AFRL-RY-WP-TR-2015-0179

SEMANTICALLY AWARE FOUNDATION ENVIRONMENT (SAFE) FOR CLEAN-SLATE DESIGN OF RESILIENT, ADAPTIVE SECURE HOSTS (CRASH)

Howard Reubenstein, Theophilos Giannakopoulos, Silviu Chiricescu, Amanda Strnad, and Joseph Fahey

BAE Systems

FEBRUARY 2016
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals.

Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2015-0179 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

//Signature//

TOD J. REINHART, Program Manager
Avionics Vulnerability Mitigation Branch
Spectrum Warfare Division

//Signature//

DAVID G. HAGSTROM, Chief
Avionics Vulnerability Mitigation Branch
Spectrum Warfare Division

//Signature//

TODD A. KASTLE, Chief
Spectrum Warfare Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
February 2016	Final	11 August 2010 – 28 September 2015

4. TITLE AND SUBTITLE
SEMANTICALLY AWARE FOUNDATION ENVIRONMENT (SAFE) FOR CLEAN-SLATE DESIGN OF RESILIENT, ADAPTIVE SECURE HOSTS (CRASH)

5a. CONTRACT NUMBER
FA8650-10-C-7090

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)
Howard Reubenstein, Theophilos Giannakopoulos, Silviu Chiricescu, Amanda Strnad, and Joseph Fahey

5d. PROJECT NUMBER
3000

5e. TASK NUMBER
RY

5f. WORK UNIT NUMBER
Y0V4

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

BAE Systems
6 New England Executive Park
Burlington, MA 01803

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

DARPA/TCTO
675 North Randolph Street
Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RYWA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RY-WP-TR-2015-0179

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. Report contains color.

14. ABSTRACT

The Semantically Aware Foundation Environment (SAFE) project provides a highly reliable, secure operating environment that substantially advances the state of the art with respect to fielding secure software systems in a hostile environment. The goal of the SAFE project, which is part of the larger DARPA Clean-slate Design of Resilient, Adaptive Secure Hosts (CRASH) program, is to create a secure, robust computing environment. SAFE takes a clean slate approach, starting with secure hardware, and then layering on formally verified software components. A key cross-cutting design goal of the SAFE computational stack is to make safety the default consideration, and to make this default (safety) easy to program. The delivered SAFE system consists of a high-fidelity hardware simulation using field programmable gate arrays (FPGAs), with a set of runtime services (ConcreteWare) running on the hardware. Secure applications can be prototyped in the Breeze high-level programming language; lower-level services are written in the Tempest systems programming language. SAFE provides a substrate upon which to build resilient applications and higher level secure languages.

15. SUBJECT TERMS
secure computing, non-interference, memory-safety, co-design

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Tod J. Reinhart
Unclassified	Unclassified	Unclassified	SAR	52	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

Table of Contents

Section **Page**

ACKNOWLEDGMENTS

The following institutions and individuals joined BAE Systems as part of our SAFE team and we would like to acknowledge their contributions to the work described herein:

Institution	*Contributors*
BAE Systems	Timothy Anderson, Edward Amsden, Raymond Bahr, Rebecca Cathey, Silviu Chiricescu, Gregory Eakman, Joseph Fahey, Michael Figueroa, Karl Fischer, Gregory Frazier, Anthony Gabrielson, Brad Galego, Theophilos Giannakopoulos, Hillary Holloway, Tom Hawkins, Siraj Iyer, Aleksey Kliger, Basil Krikeles, Marc Krull, Young Hyun (Albert) Kwon, May Leung, Bryan Loyall, Andy Macbeth, Joshua McGrath, Sumit Ray, Jothy Rosenberg, Nancy Stafford, Amanda Strnad, Greg Sullivan, Andrew Sutherland, Charles Tao, Arun Thomas, Kayvon Touran, Peter Trei, Christopher White, John Wiegley, Benjamin Wise, David Wittenberg
University of Pennsylvania	Benjamin Pierce, Andre DeHon, Jonathan Smith, Arthur Azevedo de Amorim, Delphine Demange, Udit Dhawan, Brook Fugate, Michael Greenberg, Catalin Hritcu, Ben Karel, Albert Kwon, Benoit Montagu, Robin Morisset, Nikos Vasilakis
Harvard University	Greg Morrisett, Gregory Malecha, David Pichardie, Randy Pollack, Jesse Tov
Northeastern University	Olin Shivers, Mitchell Wand
Clozure Associates	Gary Byers, Mikel Evins, Luke Palmer, Greg Pfeil, Andrew Shalit, Shannon Spires
Good Harbor Security Risk Management	Richard Clarke, Jacob Gilden, Emilian Papadopoulos, Evan Sills
Atomic Rules	Shepard Siegel
Consultants	Nicholas Citrone, Thomas Knight

1.0 SUMMARY

At the Defense Advanced Research Projects Agency (DARPA) Clean-slate design of Resilient, Adaptive Secure Hosts (CRASH) program kick-off meeting the program manager, Howie Shrobe, explained to the performers:

- The threat of malicious cyber activity was increasing and government officials have publically asserted that we would lose a cyberwar (in no small part due to our advanced dependence on computer technology).
- The work effort advantage in attack is currently strongly in favor of the attacker (the defense must defend all attack points, the offense needs to only be successful at a single point).
- Our current security strategy (in 2010 and still in 2015) of "perimeter protection, patch, and pray" is not aligned with the threat. Programmers will not bail us out of this situation (by writing defect free code).
- Software is a part of many cyber-physical systems, e.g., automobiles, weapons systems, and medical devices.
- We were entering an age where physical/kinetic war might be waged or supported via cyber-attacks (e.g., Stuxnet and the 2008 Georgian cyber campaign).

The summary of these observations was that: "the US has adopted a strategy of layering defenses on top of a uniform and vulnerable architecture. This has been necessary to buy *tactical breathing space*, but it is not convergent with the evolving threat."

The program was charged with "making the attackers push the rock" (shifting the workload against the attacker and in favor of the defense) and with developing technologies that could ultimately protect our country.

As a result of our Semantically Aware Foundation Environment (SAFE) effort under the CRASH program, we now understand how to develop inherently secure computing technologies. These technologies use computational mechanisms to remove *whole categories* of vulnerabilities – thus freeing us from the paradigm where each bug could reveal anew some vulnerability. For example, the SAFE processor is immune from buffer overflow vulnerabilities and code injection vulnerabilities. We also designed hardware support for the required technical mechanisms which grounds the SAFE platform in a trusted hardware base (based on the program premise that, unlike the 1970's, hardware is now inexpensive and some transistors can be employed solely to improve security). It has been observed that software only security solutions present two flaws: a potential increase of the attack surface and possible subversion by attacks at a lower layer in the software stack. Finally, we have verified the designs of certain crucial components of the SAFE solution, i.e., the memory safety and compartmentalization of the additional hardware mechanisms (e.g., the tag management unit and the atomic group unit) and the non-interference property of a subset of the application programming language.

Much work remains to be done and these areas will be discussed in the conclusions. While there is not an inherently secure processor available "off-the-shelf" today, the SAFE project and the CRASH program as a whole have demonstrated that inherently secure processing is feasible. As these technologies develop, we must avoid the following pitfalls: there is no wholly technical solution to cyber-security (as the computational technology improves attackers will, e.g., ramp up their social engineering and phishing attacks – all useful technology is capable of being

misused to execute inappropriate behavior); security will not come at zero cost and we must not inflict the cost expectations of mundane computing entertainment platforms (where even 10% overheads may not be acceptable) onto mission critical compute platforms (where an overhead of 50% might even be acceptable if it achieves secure processing); consumers will have an insatiable appetite for performance but in mission critical computation we must be willing to devote some hardware – that might otherwise be exploited for performance – to secure operations.

A new generation of inherently secure processor architectures is feasible today.

2.0 INTRODUCTION

Current computer host systems (hardware, operating system, and applications) have proven highly vulnerable to cyber-attacks ranging from buffer overflows, to return-oriented programming, to numerical overflows and thousands of other attack vectors (http://cwe.mitre.org/index.html). Our overall computer security posture is based on a series of "patch and pray" activities where vulnerabilities are patched as soon as feasible only to yield to new attacks in a never ending cycle.

How did our computing infrastructure end up in this posture? The Department of Defense (DOD) "Strategy for Operating in Cyberspace" (July 2011) gives a nice concise explanation:

> "The Internet was designed to be collaborative, rapidly expandable, and easily adaptable to technological innovation. **Information flow took precedence over content integrity; identity authentication was less important than connectivity.**"

Our current host computers and operating systems are of pre-Internet design and have evolved to facilitate information processing and sharing. Current technology has been designed for speed and functionality rather than security. The hardware itself treats both code and data as "raw seething bits" with no distinction between data and instruction nor any notion of provenance nor built-in access control.

Current computer security is unintentionally premised on zero defect operating system (and application) implementations. In many contemporary machine architectures any breech of intended operation is sufficient to compromise arbitrary components of the system (due to unitary privilege implementations).

Reversing this situation requires a clean-slate revisit of current computer hardware and OS architectures. Security protections must be provided that are robust against individual design faults (using defense-in-depth and least privilege design mechanisms). Designs must protect against entire classes of software weaknesses. Finally, perimeter protection mechanisms (supported by pervasive information-flow control) need to be provided that can operate correctly independent of the overall size and complexity of the code inside the perimeter.

The goal of the SAFE (Semantically Aware Foundation Environment) project, which is part of the larger DARPA CRASH (Clean-slate design of Resilient, Adaptive Secure Hosts) program, is to create a secure, robust computing environment. As part of CRASH, SAFE takes a clean slate approach, starting with secure hardware, and then layering on formally verified software components. A key cross-cutting design goal of the SAFE computational stack is to make safety the default consideration, and to make this default (safety) easy to program.

A simplified model of the concreteware (low level system software) has been verified correct with respect to security properties such as memory safety and compartmentalization and the design has been targeted to provide non-interference (though language constructs have been added that permit necessary audited/intended violation of non-interference). The SAFE project provides a highly reliable, secure operating environment that substantially advances the state of the art with respect to fielding secure software systems in a hostile environment.

The delivered SAFE system consists of a high fidelity hardware simulation hosted on a Field Programmable Gate Array (FPGA), with a set of runtime services (concreteware) running on the hardware. Secure applications can be prototyped in the Breeze high level programming

language; lower level services are written in the Tempest systems programming language. SAFE provides a substrate upon which to build resilient applications and higher level secure languages.

There are many existing proposed security solutions currently being deployed such as anti-virus and intrusions detection systems, address space layout randomizers, and software diversification engines. Ultimately most (if not all) of these solutions are simply exercises in making penetration harder (but not impossible) to achieve -- as any software-based solution ultimately builds on other layers of unprotected software. To avoid security solutions that simply amount to building on a "house-of-cards" there must be a base of trust which, in the case of SAFE, is the hardware implementation using a security tag management unit (TMU). Evidence of the futility of many of these clever "house-of-cards" software solutions comes in the form of the "Blind, Return, Oriented, Programming (BROP)" work out of Stanford University (http://www.scs.stanford.edu/brop/). While this work does not discredit all security solutions it does illustrate how little information is required to compromise a system (in this case they compromise a closed-binary and source code component needing only the ability to overflow the stack and a process to respawn the component after it crashed).

Using the SAFE host computer, the programmer is building upon hardware that directly enforces security guarantees with respect to information flow, typed values, and buffer bounds checks. The operating system (OS) of the SAFE system is thus provably secure against whole classes of attacks (e.g., buffer overflows and code injections).

Applications can be designed to be secure using the same mechanisms, but even if they do not use these mechanisms then successful attacks are only possible on a per application basis instead of against the common operating system. In this way, the workload is shifted away from favoring the attacker and back to favoring defenders of the system (by eliminating high value attacks against the common OS).

Any useful program is going to need to do some dangerous things, e.g., exfiltrate sensitive data. In SAFE all security policies are enforced by default. There are explicit operators that can perform dangerous activities such as declassifying data to lower secrecy levels or even to public accessibility. Thus exceptions to policies are localized and can be reviewed and audited. Importantly, the cost of such reviews does NOT increase proportionally with code size. Instead reviews require inspection only of the explicitly coded exceptions. The review allows determining exactly those conditions under which it is appropriate for a secure application to perform risky operations.

For the reader: This report provides guidance to the issues faced in the design of our clean-slate secure host computing solution and a summary of the results. Detailed programming guidance is provided in technical reports associated with the project. This report should prove useful to anyone contemplating an effort to create a secure system level platform particularly in terms of scoping the effort and appreciating the issues that will be faced in designing a comprehensive secure solution. Section 3.0 focuses on the overall scope and assumptions behind the SAFE effort. Section 4.0 provides a subsection-by-subsection description of the primary results of the project. Section 5.0 frames conclusions in terms of lessons learned and unresolved issues.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

The SAFE processor was developed using agile software and hardware development methods (though the hardware was designed using a software modeling language called BlueSpec so that the main hardware specific methodology involved reducing the design to an FPGA implementation). The primary aspect of the development method that was exceptional in the SAFE project was our emphasis on co-design. Whereas some project efforts might coordinate efforts via early development of application programming interfaces (APIs) and subsequent configuration control, co-design involved extensive communication and feedback between component development teams and the willingness to modify designs on a semi-continual basis to satisfy newly emerging requirements. Co-design avoids early commitment to design decisions (made without full understanding of the context in which these decisions will operate) at the cost of significant communication overhead and multiple co-dependent development iterations.

The assumptions that provide the basis for understanding the results of the project effort revolve around the overall high-level architecture and design principles that were established. This section reviews those decisions and the individual components and tools of the SAFE project are described in more detail in the results section. An important tenet of the project was that secure operation of the SAFE machine, which is provided by a few fundamental protection mechanisms, is grounded in hardware support.

The fundamental SAFE protection mechanisms are:

- Memory safety supported in the hardware by unforgeable “fat” opaque pointers. This protection eliminates buffer overflow vulnerabilities and related memory attacks.
- Hardware enforced data types supported by an atomic group mechanism. This supports distinguishing between: data, pointers, and instructions. Among other vulnerabilities this protects against code injection.
- Least privilege enforcement supported by an authority creation and passing mechanism. This supports organization of the OS and applications into least privilege containers eliminating root escalation vulnerabilities.
- Perimeter data protection (information flow control and label models) supported by the rules in the Tag Management Unit (TMU) and carefully design system Input/Output (I/O) channels. This provides data access control that can eliminate data exfiltration vulnerabilities.

SAFE memory is organized as a memory space of **atoms**. A SAFE memory atom is an indivisible unit of storage that contains: a payload (the traditional data stored in memory) and a tag (meta-data about the payload). The tag consists of an atomic group (a hardware understood type for the data) and a tag pointer (to metadata which itself is stored as atoms in memory). Every payload item is tagged with a full pointer, unlike many previous architectures that assign only a small number of fixed bits. SAFE supports arbitrary metadata. In general the ALU processes the payload and the TMU processes the tag. There is very limited and privileged crossover of data. For example, the TMU miss handler does have to transfer tag data into payload so that it can compute the result of the tag computation. Setting the resulting tag is an operation privileged exclusively to the TMU. User applications cannot directly set tag data.

These mechanisms are provided by an architecture that consists of:

- SAFE processor (simulator and FPGA)

The SAFE processor consists of a number of architectural additions (listed and described below) to a standard processor architecture. An important security design goal of the overall SAFE machine is that there is no shared memory between processing threads. This provides for memory protection between threads. Thread-to-thread communication is supported by copying data over channels without sharing addresses.

The SAFE runtime is fully distributed. Each user thread operates in its own address space (which unlike traditional systems is protected at the hardware and software level as an effectively disjoint memory space due to allocation as a bounded memory frame and the memory protection afforded by opaque, "fat" pointers). The principals, authorities and tags (PAT) server provides a separate address space for principles, authorities and tags (which must be shared across threads). Operating system services run locally (per thread) and are accessed through a secure gate invocation mechanism. Global services (e.g., PAT server or memory manager) run in threads that are accessed via streams/channels.

- Tag Management Unit (TMU) and TMU Cache and Atomic Group Unit (AGU)

The TMU operates on every instruction cycle and checks to see if the proposed operations obey the set of security rules installed in the TMU. The input to the TMU is an "M vector" consisting of 9 fields that describe the tags on the inputs to the operation. The output of the TMU is an "R vector" consisting of 8 fields that update the tags on the data being operated and a Boolean field declaring if the operation should be allowed.

The TMU operates using parallel hardware paths to the normal arithmetic logic unit data paths. Separation of the data paths provides for secure operations but nevertheless the two must run in synchronization. To address efficiency issues of the TMU processing, a TMU cache is introduced to provide efficient lookup of already computed <M,R> vector TMU computations. Performance data for this mechanism is described in (Dhawan, 2014)

The TMU implements a number of different security oriented policies. These include: information-flow control based on label models, data signing and sealing, low-level type safety (enforced by the atomic group unit), memory safety (via fat pointers), and linearity.

Policies are implemented as a set of symbolic rules referred to as label models. The SAFE system combines multiple defined label models into an overall product label model. Specific label models often focus on a subset of instructions that are governed by that label model.

- Fat Pointer Unit

Fat pointers are used to encode the base and bounds of all pointers. This allows the hardware to avoid out-of-bounds memory reads and write. The mechanism is implemented in an efficient "low-fat" algorithm which optimizes encoding to reduce overhead (documented in (Kwon, Dhawan, Smith, & Knight, Jr., 2013))

Pointers to memory frames are provided as opaque data structures. Arbitrary pointers cannot be created from, e.g., integers and are only created anew by a privileged memory allocation authority.

- Garbage Collector (GC)

Memory management in SAFE is supported by a contemporary garbage collection scheme. While portions of the underlying SAFE OS will use explicit memory management, overall applications are secured by relying on a correct (and verifiable) garbage collection implementation.

The SAFE Garbage Collector uses a thread concurrent copying approach. In a concurrent approach, garbage collection takes place on a per thread basis, rather than a system stop-the-world collection in which all useful work stops while garbage collection takes place (though each thread uses a stop-the-thread GC scheme). Copying means that rather than try to leave data in place, and put new data where the dead data used to be in small chunks, we copy the "live" data out of one space and into another, and then re-use the entire space.

There are three regions of memory: old-space, copy-space, and new-space. Allocation is done from new-space, while any live frames from old-space are copied to copy-space. The GC maintains the invariant that nothing in new-space points into old-space by copying anything from old space when it is written. When all the live frames from old-space have been transferred to copy-space, old-space can be re-used. The GC only copies complete frames.

- Scheduler

The scheduler currently implemented in SAFE is a pre-emptive round robin scheduler. This means that each thread is allowed to run for a maximum set length of time, after which it is interrupted and the next thread is allowed to run. Threads are run one by one, always in the same order. After all threads have been run, the first thread is run again.

- PAT Server

The PAT server provides a shared space of principals, authorities, and tags to support secure shared TMU computations across threads. The PAT server supports the definition of label models and rules and uses these to compute TMU miss computations in a server that can be off-boarded to a separate component in the distributed SAFE operating system.

The SAFE machine makes use of three programming languages: an assembly/ISA machine level programming language, a system level programming language (C-like) called Tempest, and a prototype functional application language called Breeze.

- Instruction Set Architecture (ISA)

The SAFE machine language was designed concurrently with the Breeze application language. It offers hardware support for many of the complex parts of the high level language. Thus, along with many of the typical assembly instructions such as ADD and YIELD, there are others such as ARETAG and BCALL to support data tagging and secure invocation. As more was learned about the label models, instructions were updated to take into account any vulnerabilities discovered. Thus the SAFE ISA is not simply a typical control and arithmetic instruction set as it includes categories of operation for: authority management, TMU management, tag management, and secure stream management.

- Tempest

Tempest is a system level programming language with lower-level features than Breeze. Tempest is a typed language that supports system-level explicit memory management, explicit calling conventions, inline assembly insertion, and linear (non-copyable and thus non-shared)

pointers. Nevertheless most Tempest applications are garbage collected. Explicit memory management is performed by OS services supporting the memory manager and the stream manager (which must allocate from global memory to support inter-thread communication).

Tempest's support for linear pointers includes language features for easing the use of the pointers where copying would occur in traditional compilers (e.g., explicit register spilling support, as well as features for non-destructively "focusing" on components of data structures that involve linearity).

- Breeze

Breeze is a typed (mostly functional) language that supports contracts on procedures. It provides explicit syntax and libraries for programming with user-defined label models to support fine grained (instruction-level) information-flow control. It also provides a "bracketing" construct which is used to secure information flow from conditional computations. Breeze supports a non-interference model of programming whereby "sensitive" (classified or high labeled) data cannot create observable differences in public (not sensitive or low) outputs.

One of the innovations in Breeze is to avoid the use of exceptions (which are a control flow mechanism difficult to secure with IFC) and to replace them with a specific exception value, called NaV (Not a Value). This allows for a straight-forward way to express erroneous behavior without introducing potential information flow channels that come with alternative control flow mechanisms, such as exceptions. Included in the erroneous behaviors are computations that would leak information past a secrecy boundary, either directly in a value or indirectly via control flow manipulation. Attempting to do so will only yield a NaV; no secret information will be exposed. Many of the subtle cases of these attacks are described in: (Hriţcu, Greenberg, Karel, Pierce, & Morrisett, 2013)

Breeze supports least privilege program design via creation of unforgeable principals and authorities. Using these, the developer can create as many compartments as appropriate for a component or subsystem. The authority mechanism can be used to implement secure programming patters such as "propose and verify."

Separation of privilege via authority creation is supported by: authority carrying invocation gates acting as capabilities and separation of authorities between processes with constrained communication over channels that have IFC policy protections enforced by the underlying system.

3.1 Information Flow Control and Label Models

A central technique guiding the overall SAFE design is the use of information flow control processing. Information Flow Control support is at the heart of the SAFE processor design and is one of the features that distinguishes the SAFE machine from standard computer architectures. Support for information flow control pervades the design of SAFE from hardware up to application design.

Information Flow Control (IFC) is a method for tracking how data explicitly and implicitly influences the results of computation. In dynamic IFC, this is done at runtime by attaching metadata called labels to data and propagating those labels alongside the data during computation. Data explicitly influences results when it is directly combined with other data. For

example, appending a paragraph marked “secret” to a document would cause the augmented document to be marked “secret” as well. Data indirectly influences results when it causes a change in control flow that then produces some result. When IFC is used for enforcing secrecy, tracking indirect influence is necessary to prevent the control flow of a program from being used as a means of declassification. For example, branching on whether a secret string contains a particular word and then returning “true” or “false” would cause those return values to also be labeled as secret. If the results were not labeled secret, information about the contents of the string would be revealed.

Fine-grained IFC involves applying this method at the level of the individual operations on data. On the SAFE system, this means applying various *label models* to each instruction as it executes. The hardware and an OS component collaborate to enforce these label models on every other part of the system, including other OS components.

On the SAFE system, the *secrecy* label model is responsible for ensuring that secret data does not undergo exfiltration. The secrecy label model and its dual the *integrity* label model are described in (Montagu, Pierce, & Pollack, 2013). The particular incarnation of the secrecy label model that is used on SAFE is the one described in “Disjunction Category Labels” (Stefan, Russo, Mazières, & Mitchell, 2012). Instead of a full integrity label model *a la* Disjunction Category (DC) integrity, SAFE makes use of a *signature* label model, the use of which resembles the use of cryptographic signatures. The signature label model is the integrity portion of the *sealing* label model that appears in “Micro-Policies: Formally-Verified, Tag-Based Security Monitors” (Azevedo de Amorim, et al., 2015). “Micro-Policies” also describes several other label models, such as memory safety and atomic groups (i.e. hardware types), that could be implemented in the SAFE system if there was not built-in hardware support for enforcing the properties that the label models are intended to enforce. For an in-depth discussion of those policies as label models and the verification efforts overall, see Section 4.6.

DC secrecy and DC integrity are both non-interference label models. The non-interference property is that high inputs (secret or low-integrity data) cannot influence low outputs (public or high-integrity data). Because the SAFE system supports explicit declassification operations and communication between concurrently running processes, true non-interference is impossible. However, the label models are implemented in such a way as to provide non-interference within a single process in the absence of declassification operations.

There are several other implementations of IFC through the use of label models. Jif (Java information flow) is a system for utilizing IFC on the Java Virtual Machine (Myers & Liskov, 2000). Jif also adds support for groups by allowing principals to act on behalf of each other. Asbestos and HiStar are alternative label models that apply labels more coarsely than SAFE using a mapping of principals (as ownership) to security "levels” (Efstathopoulos, Krohn, VanDeBogart, Frey, et. al. 2005; Zeldovich, Boyd-Wickizer, Kohler, & Mazieres, 2006). Labeled IO (LIO) offers an implementation embedded in the Haskell programming language (Stefan, Russo, Mitchell, & Mazieres, 2011). Hails builds on LIO and DC labels to provide an embedded haskell framework for developing secure web applications (Griffin, Levy, Stefan, Terei, et. al. 2012). SAFE, however, is the only system to offer fine-grained IFC with hardware support.

As described in “A Theory of Information-Flow Labels” (Montagu, Pierce, & Pollack, 2013), the labels used in SAFE form a lattice-based algebra. DC secrecy has the public label (anyone can read) at the bottom of the lattice and completely private (no one can read) at the top. Our signature label model has the label indicating a value has not been signed at the bottom, and a label indicating that it has been signed by every principle at the top. SAFE combines these two lattices into a product label model, with the resulting lattice as the product of the two lattices. Operations cause labels in different components to propagate differently, according to each label model’s policy implementation. In particular, while secrecy labels flow to all data that was influenced by the secret, signature labels (unlike DC integrity labels) only flow when data is exactly copied or is moved. For more information on the implementation of label models in the SAFE system, see Section 4.3 on the SAFE OS’s label model enforcement mechanism.

Label models in SAFE are augmented by a clearance mechanism that provides something akin to access control. This clearance mechanism sets an upper bound on the label of the data that can influence control flow in a process, providing a way to mitigate the ability for malicious programs to use communication between concurrent processes to reveal secret data.

3.2 SAFE Platform Attack Model

We assume a correctly implemented instruction set architecture and supporting hardware. We further assume physical security, the absence of hardware-layer tampering or supply-chain attacks. However, we do assume an attacker can obtain privileges on the machine comparable to any user. While we provide compiled languages we do not assume the attackers must use the compiler. Our model assumes the attacker can author assembly code as a direct proxy for ISA machine code. The fundamental security question is then whether the attacker is fully-isolated on the machine unable to read or write protected data of other users and unable to inject computations into another user’s thread space.

4.0 RESULTS AND DISCUSSION

Instantiation of the SAFE architecture resulted in advances in each of the high-level architectural features discussed in the previous section. This section discusses in more depth the results in the areas of:

- The overall *tagged hardware machine* and supporting *ISA* (which was ultimately reduced to an FPGA-based reference and demonstration machine).
- Two high-level *programming languages*: *Breeze* used to develop applications and *Tempest* used to develop the machine operating system. We also discuss the supporting tool environment.
- The *SAFE operating system* referred to as Concreteware.
- Two high level *demonstrations*: a database information protection demonstration called *SAFE Knowledge Online (SKO)*, and a control-system integrity demonstration (*rocket controller*); and one low-level information-flow control demonstration (*sum server*).
- The SAFE development tooling environment (*SAFE Tools*) supporting both execution simulation and FPGA-based execution.
- An annotated bibliography of the detailed published results in the *verification* of SAFE computational models.
- An annotated bibliography of the detailed *hardware design* mechanisms and efficiency/costs of the SAFE tagged memory processor.

4.1 Tagged Hardware Machine

The fundamental compute "word" in the SAFE architecture is the atom (see Figure 1). An atom is an indivisible quantity that consists of three parts: an atomic group (AG), a tag and a payload. The atomic group describes the general type of the data - an Instruction Pointer, an Integer, an Instruction, etc. The tag field holds a pointer to the metadata for this atom. The payload is the actual data. In our implementation, the atom is 128 bit wide, with the payload, tag, and AG being 64, 59, and 5 bits respectively. All the memory and the architectural state are represented in terms of atoms. The payload part of the atom is the actual data from a traditional architecture. The SAFE machine supports the same type of instructions traditionally found in any RISC architecture – arithmetic, logic, control, load/store, etc. Those instructions work on the payload part of the atom in the same way as expected; however, in parallel with the operations on the payload, the SAFE machine also works on the metadata part of the atom (tag and AG). It is only the hardware that splits the atom into its constituent parts and performs the computations on the tag and payload in parallel. Once the computation is completed, the hardware assembles the AG, tag and the payload back into an atom and writes back the results to its destination as an atom.

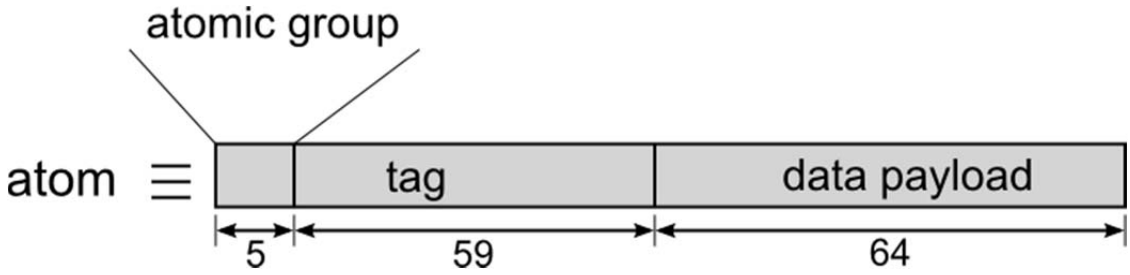

Figure 1: Atom Encoding

The ISA also contains tagged architecture-specific instructions that work on tag management (i.e. retag, aretag), TMU management (i.e tmul, tmurc, etc), authority management (i.e. inp, ina, etc.), atomic group management (i.e. ingrp, regrp), and secure invocation of operations (i.e. bcall, acall, gacall, etc.). These "non-traditional" instructions have varying levels of restrictions (i.e. can be executed only in some special privileged mode – that is, the instruction itself must be tagged with a special tag.).

Another set of instructions that work a bit different than the "traditional" RISC instructions are the memory-related instructions; they are different in the sense that they operate on "fat pointers" – which encode both the base and bounds (Kwon, Dhawan, Smith, & Knight, Jr., 2013); as with all the tagged architecture instructions, the tag portion of the atom is also set on the destination of operations such as memory read/write/copy/mv. Arithmetic operations on fat pointers are strictly forbidden. There are however a set of operations (e.g., pointer offset operations) that work directly on pointers but these operations are privileged (e.g., only the GC is allowed to use them) and enforce memory frame safety.

A good description of the implementation of the SAFE architecture can be found in (Chiricescu, et al., 2013); the following are some of the most "non-conventional" hardware blocks that were present in the implementation and which are shown pictorially in Figure 2

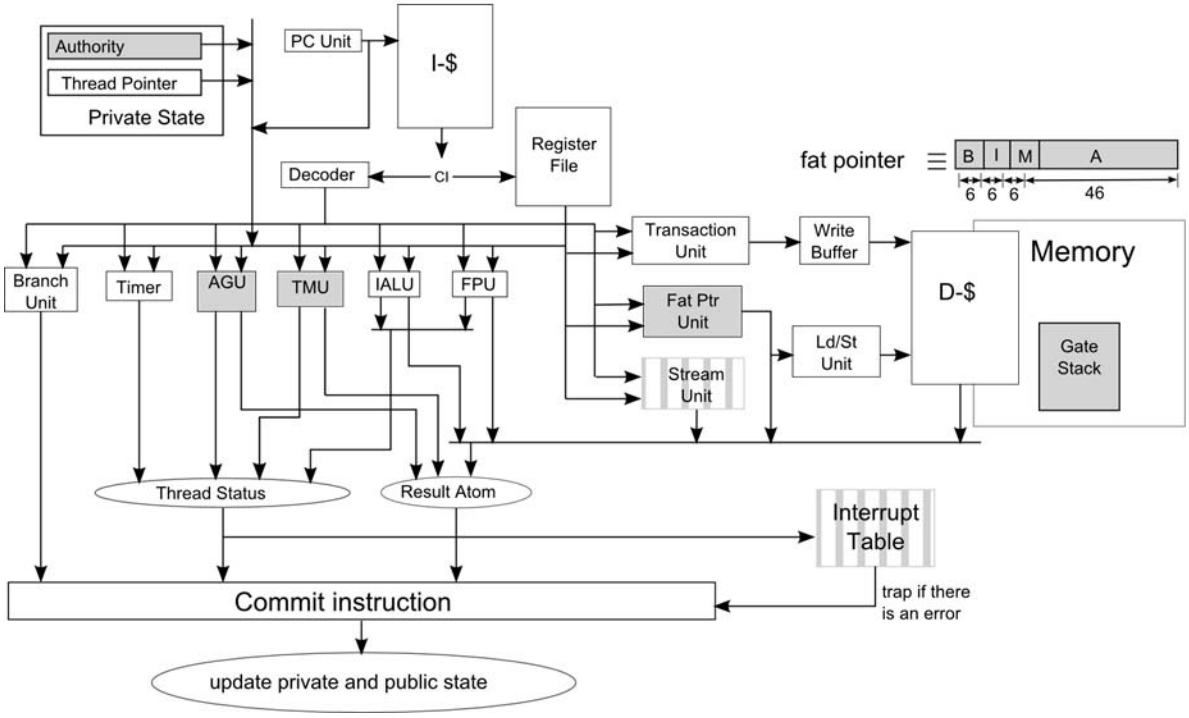

Figure 2: The micro-architecture of the SAFE processor

- Atomic Group Unit (AGU) - responsible for checking that the atomic groups of the current instruction operands match those required (e.g. an arithmetic operation requires that both operands are of type Integer);There are 19 atomic groups defined in the SAFE architecture.
- Tag Management Unit (TMU) – responsible for checking (for every instruction) that the metadata for each atom involved (program counter (PC), instruction, and each operand) is evaluated against an installed ruleset and either an access violation is flagged or the metadata for the result is returned. To ensure no pipeline stalls, this check must happen in parallel with the rest of the machine execution and should take no more than 1 cycle; thus, a fast near-associative hash cache is being used (Dhawan & DeHon, 2013);
- Fat Pointer Unit – this unit manages the "low-fat pointers" (Kwon, Dhawan, Smith, & Knight, Jr., 2013) which encode the base and bounds of a pointer. To avoid performance degradation due to complex decoding, the decoding has been split up into two "stages";
- Gates – hardware support for fine grain domain crossing (e.g. switching from user mode to kernel mode); a gate is similar to closure (in functional programming) as a pointer to the gates local storage is maintained. The current authority (e.g. 1st class mechanisms for isolating privileges) and local storage pointer are pushed onto a gate stack when a gate call is being made; the authority register is populated with the target gate's authority, and the target gate's local storage pointer is installed. The gate stack is not accessible to any other gate call or return instructions;

4.1.1 SAFE ISA

To a certain extent, the SAFE ISA was influenced by the development of Breeze. It was intended to offer hardware support for many of the complex parts of the Breeze high level language. Thus, along with many of the typical ISA instructions such as ADD and YIELD, there are others such as ARETAG and BCALL – which have direct correspondence in the Breeze language. As more was learned about the label models, instructions were updated to take into account any vulnerabilities discovered. In general, instructions that were no longer needed were only removed if they created a security hole. Thus some instructions remain that are no longer particularly useful operations

As mentioned before, the AG specifies the type of data that is stored in the payload – an Integer, an Instruction, a Pointer. On every instruction, the machine checks to ensure that only the correct type of data is allowed. For example, the different types of pointers are considered completely different atomic groups. While a ThreadPointer and a FramePointer may both point to something that looks like a thread frame, only the ThreadPointer can be used to run that thread. Similarly, an Integer that has the same payload as that ThreadPointer cannot be used to run the thread. SAFE also prevents arithmetic operations on pointers. While it is possible to convert a pointer to an integer value and then modify it, it is not possible to go from an integer value to a pointer. This makes forging pointers impossible. SAFE also has a less common type of pointer, called a LinearPointer. LinearPointers cannot be copied, though they can otherwise be used like other pointers. The various atomic groups are enumerated and described in (DeHon, Dhawan, & Strnad, 2014).

Besides the "typical" RISC instructions, the SAFE machine contains a number of instructions that have been implemented to provide security. These instructions can be grouped into:

1. Authority management – instructions that deal with manipulations of first class authorities and principals; some of these instructions are privileged.
2. Tag management – these instructions are used for the creation and manipulation of tags. They are used by user code to access first-class tags for the purpose of checking their algebraic properties (such as asking the PAT server whether one tag is more secret than another) and by the TMU miss handler, which converts tags to and from pointers to the tag metadata on which the PAT server operates.
3. TMU management – these instructions are privileged and used only by the TMU handler to update the TMU or gate call caches or to read performance counters.
4. Group management. – these instructions are used to inspect the atomic group (e.g. convert it into an integer), or to change the AG of an existing atom (restricted to certain parts of the OS).

Another interesting set of instructions are related to the inter-frame control flow – particularly the gate calls and bracket calls. A gate call is essentially the equivalent of a closure that can be executed with different authorities (e.g. implicit, specified explicitly by the caller, augmented, cacheable). Gate call creation and return from gate calls round up the gate-related inter-frame control flow instructions. A bracket is another inter-frame control flow mechanism influenced by Breeze, and it is being used as a replacement for automatic PC tag lowering and as a mechanism for avoiding poison pill attacks (Hriţcu, Greenberg, Karel, Pierce, & Morrisett, 2013). As with

the gate calls, there are a few "flavors" of bracket calls available: with implicit authority, with authority change/augmentation, etc.

4.2 Programming Languages

The overarching design goals of the languages for the SAFE system were

1. to be useful for exploring the impact of label models on secure OS and application development,
2. to allow exploration of the impact of the other safety mechanisms (linear values, gates, bracket calls) of the SAFE system on secure OS and application development, and
3. to construct a working, secure, minimal operating system, according to the principle of least privilege.

4.2.1 Breeze

Breeze is a high-level, dynamically typed, mostly functional language with support for concurrency, channel-based communication, and fine-grained information flow control policies (i.e. label models). The purpose of Breeze was to be able to explore the impact of the enforcement of the secrecy and integrity label models on applications early in the program. The current implementation of Breeze is as an interpreted language running on traditional systems. There is a partial implementation of a Breeze-to-SAFE compiler, which was intended to allow our prototype Breeze applications to run on the SAFE hardware.

Breeze's IFC implementation uses secrecy and integrity non-interference label models in the style of DC labels. The runtime IFC implementation is mostly standard: primitive operations on data cause labels to flow to results, and control flow operations on data cause the label on the data to flow to an ambient "PC" (program counter) label. The PC label is then used to restrict whether channel communication may occur. Additionally, processes execute under a clearance, which is a label that acts as an upper bound on the PC label. This mechanism allows for some protection against covert channels.

In order to enable the limited exfiltration of secret data, Breeze programs also have a notion of authority, with which a program can lower the label on a value—either reducing its level of secrecy or endorsing it to increase its integrity. Authorities in Breeze are first-class values that can be used to implement capabilities by capturing them in closures that perform specific actions when applied. Authorities can also be registered as ambient, allowing called code to act with a given authority, but not allowing it to hand that authority to another process via a channel.

A *poison pill* attack is when a malicious agent provides a process with some value that it cannot use (e.g. due to the clearance of the process). In order to avoid such attacks, Breeze provides a mechanism to acquire and inspect the labels on values, and requires that labels acquired in this way are public information to all code. This design decision created a need for a mechanism to apply labels to values that would not leak information about the data via the labels. The novel mechanism that Breeze uses for this purpose is called a *bracket*.

Brackets serve two purposes: to apply labels to values and to restore the PC label. The key property of using brackets is that the resulting label upon exiting the bracket must be chosen **before** entering the bracket (and thus no information can be leaked from the bracketed computation which has not even been executed yet).

The label that may be applied by a bracket is bounded below by the PC. Since there is no way to lower the PC label without authority other than brackets, the label to be applied to a value must be chosen before any processing of secret data occurs. In the following example, the PC label starts at a low secrecy label `L`, enters a bracket where it inspects a value with a high secrecy label `H`, and then returns a value that is labeled with the pre-chosen label from the bracket and has the PC restored to `L`.

```
{ if secretValue@H then 1@L else 0@L }@H
```

Brackets allow a user to choose when to move the protection on a value between the PC label and the label on the value itself. In the above example, before exiting the bracket, the label on the resulting `1` or `0` is still low. The choice of value is protected by the PC. A programmer could choose to do more operations with the high PC label protecting the values in scope before finally choosing to move the label from the PC to the value that will be used by the rest of the program.

To avoid the complexities that would be caused by the extra control flow possibilities, Breeze does not have exceptions. Instead, errors in Breeze—including those caused by label model violations—are reified into first-class entities called NaVs (Not-a-Values). NaVs can be passed to and returned from functions and stored in data structures. For an in-depth discussion of NaVs, see "All your IFCException are belong to us" (Hriţcu, Greenberg, Karel, Pierce, & Morrisett, 2013).

4.2.1.1 Architecture of Applications in Breeze

In order to design a secure program in Breeze, a developer must first consider the labeling on sensitive information the application handles, such as passwords, financial data, or classified information. Unlike traditional software development using imperative programming languages, information flow control is the primary concern during the designing process for an application. Through this process, a developer needs to identify what the sensitive information is, what privileges are needed to operate on the data, and lastly where the privileges are needed in the application. In Breeze, authorities and their associated principals are used to label and operate on sensitive information.

Like most modern programming languages and frameworks, Breeze application development typically emphasizes the concept of modularity. Modularity is important because a major goal of secure applications in Breeze is to allow mutually distrustful components to interact with each other and exchange sensitive information. In addition, components need to independently manage privileges.

When designing an application or component in Breeze, a developer strives to identify an appropriate separation of privileges and tasking. By strategically dividing a component into subcomponents, a subset of the privileges may be distributed amongst them. Through this approach, each subcomponent may be run with fewer privileges to increase compartmentalization and modularity across the system.

Each compartmentalized component in a Breeze application is responsible for a small number of privileged operations and its own principal(s). A developer ensures that the authorities associated with the component's principal(s) are locally bound so that each component has the ability to control access and declassification to its own private data without worrying about other components' behavior.

A common example to explore the separation of privileges and labeling in IFC is a tax preparer application. Client 'Bob' would like to have his taxes prepared by the 'TaxPreparer' application. In this example, Bob's financial data is sensitive information that he would not like leaked for the world to see and does not want to simply 'trust' the application with his data. In addition, the tax preparer does not want to share the proprietary source code of its application with Bob to gain his trust. From an IFC standpoint, Bob has privilege 'B', and the tax preparer has privilege 'T'. While both B and T's privileges to declassify are kept separate from one another, Bob can label his financial data such that the tax preparer may only access (read) and make computations on his data. Through IFC, the resulting tax information will also only be declassified with the use of Bob's personal, private privilege. In Breeze, this may simply look like:

```
taxData = {
  raiseClrBy Bclr B;
  raisePcBy B;
  TaxPreparer.prepare( bobsFinances @ B );
} @ B;
```

In this example code, only Bob's read privilege is needed from Bob for the tax preparer to do its job. Because a bracket surrounds the function call, all products of the computations will also be private to Bob. The TaxPreparer application could not send any of Bob's information into the public because such a flow of information would require Bob's declassification privilege, which it does not have direct access to. Designing a secure application in Breeze first requires a developer to identify an appropriate separation of privileges and logic into mutually distrustful components.

Modularity in Breeze application design is also commonly supported through the use of threads and channels. While invoking functions, such as in the example above, is an easy way for components to interact, greater compartmentalization can be achieved by providing each component its own thread and requiring communication to occur over channels. Channels in Breeze use a label to manage the flow of information across them and can be an extremely useful mechanism for designing the way components interact in a secure application. In the tax preparer example, both Bob and the TaxPreparer application could operate in separate threads with a single channel labeled 'B' connecting the two. By using this approach, a developer can avoid accidental mislabeling on brackets or function calls and still be confident that it will not leak. Because channels automatically label the data on it with the label of the channel, even the financial data itself could be mislabeled and the channel will still protect against information leakage. Overall, use of separate component threads and communication over channels provides greater resilience to programming errors and is easier to audit. Further discussion of applications implementing channels for communication is available in the following section.

Designing and applying security policies becomes easier when an application's architecture adheres to a component per thread strategy. It becomes easier to reason about the way information flows between mutually distrustful components when channels are the primary medium for communication. A security policy can be more easily applied to an application by distributing privileges to threads and labeling the communication channels accordingly. Such an

approach also helps to give others a convenient overview of an applications components and information flow from an auditing standpoint as well.

4.2.1.2 Implementing Applications in Breeze

A number of helpful Breeze programming idioms and practices have been identified over the course of this program to better support writing secure applications. The following is a list of the more notable practices used in Breeze application development.

Writing Clean Code:

After reading through the Breeze manual, it is easy to see that application code can quickly become dominated by handling IFC functionality, brackets, and other commands specific to Breeze labels or IFC. In an attempt to help keep code readable and maintainable, it is important to look for ways in which IFC related code can be abstracted away from the application logic.

A good way to scaffold an IFC application is to begin by laying out the functionality of each application component without considering IFC. Once the general architecture is laid out, the necessary IFC functionality can be integrated through the proper abstractions. The multi-threaded sum server tutorial in the Breeze manual tackles this scaffolding process. In addition, maintaining useful abstractions for IFC operations can greatly increase the ability of a developer to debug an application.

One useful way to abstract IFC operations in a component is to create a helper function responsible for executing a generalized function, or *thunk*, with the component's authority. Consider component 'A' needing to execute a thunk with its own sensitive data:

```
let {| prin = A; clrAuth = AClr; |} = newPrin "ComponentA";
ALabel = secrecyLabel [[A]];
fun doForA thunk = {
   raiseClrBy AClr ALabel;
   raisePcBy ALabel;
   thunk();
} @ ALabel;
```

While this is a relatively simple abstraction (and may not appear to be all that necessary), when using more complex IFC functionality, such as with groups, these abstractions become much more useful for the programmer. Another helpful abstraction may involve invoking a thunk with a lower PC similar to the example above. When employing these functions in component logic, the code becomes much easier to read and also helps avoid simple programming errors that can be made by the developer.

Thinking Relatively:

Successful IFC abstractions will rarely hard-code the use of a public label. While public is the current default PC label in an executing Breeze program, it is not necessarily the PC label present when a function or component is invoked. Depending on the context, a function should treat its ambient PC classification level parametrically. When writing IFC code, a developer should try to think relatively about the context in which code will be run. The clearance and PC can be determined easily using the commands:

```
getClr ();
getPc ();
```

In addition, the function `labelOf()` can help determine the label of data, such as function arguments. Using these commands is a good way to ensure that IFC code is robust to varying conditions when invoked. Brackets are useful for automatically restoring clearance and pc upon leaving the bracket. Breeze provides a number of IFC helper functions using the above commands to assist developers to build robust code.

Sending Data Between Mutually Distrustful Components:

As mentioned in the previous section, 4.2.1.1, a necessary facet of least privilege design is the ability to send secrets between mutually distrustful components. A component may not necessarily be malicious (such as a virus), to warrant mutual distrust – in fact, honest programming errors are a far more common reason for taking this standpoint.

Commonly, a component will want to send a secret with the assurance that the secret cannot be declassified and exported; a concept approaching the idea of read-only access. A distrustful application should be allowed access to read and use a component's secret data without being able to declassify it or any working products. This can be achieved in Breeze through the use of careful labeling and/or bracketing.

Suppose two mutually distrustful components exist, A and B, each possessing the principals `aP` and `bP` respectively. Component A maintains a secret integer (43) it wishes to share with component B to carry out a computation. Labeling the secret integer `public` would be a poor idea because it wouldn't be secret. Labeling the secret with label `bLab` would also be a poor idea because it would easily allow B to declassify the secret. Labeling the secret with `aLab`, however, is a much safer alternative. Component A can confidently invoke Component B's function `computeSum` through the following steps:

```
aLab = secrecyLabel [[aP]];
bLab = secrecyLabel [[bP]];

secretNum = 43 @ aLab;

raiseClrBy aClr aLab;
result = ComponentB.computeSum secretNum;
```

In this example, A is providing B with the ambient use of A's clearance (`aClr`) to grant access to its private data. So long as A keeps its declassification authority locally bounded, it has the assurance that B cannot declassify or export `secretNum` or any computations involving the secret. Through the use of this labeling and invocation with raised clearance, A is allowing B to read and use A's secret data without declassification privileges.

Now consider the altered example below involving a bracket.

```
result = {
   raiseClrBy aClr aLab;
   raisePcBy aLab;
   ComponentB.computeSum secretNum;
} @ aLab;
```

This example follows the same behavior as the previous one, with the added constraint that B will be enforced to operate under a minimum pc label at `aLab`. By bracketing the computation, A is allowing B to read and use A's secret data with the confidence that all working products developed by B will be at a minimum label of `aLab`. The bracketed computation taints the entire computation by the label on the pc, which can sometimes make reasoning about computational flow much easier.

A restriction imposed by using a bracket in this case is the fact that B is limited to the use of ref cells and channels with a minimum label of `aLab`. In the previous example, this is not true. While this is a greater restriction on the functionality available to B, it can be seen either as a benefit or a hindrance in application design. In cases where A is certain that communication over these mediums is not necessary, a bracket may be useful as an added measure for reasoning about the flow of its secrets. In other instances where B may require use of channels or ref cells, such as in multi-threaded application design, this behavior may be crippling. It is important, however, to understand the impact of using a bracket in communication between mutually distrustful applications and decide when each should be used in Breeze application development.

Consider an update to the code above where `secretNum` is relabeled to possess the label `aLab 'join' bLab`:

```
result = {
   raiseClrBy aClr aLab;
   raisePcBy aLab;
   sharedSecret = secretNum @ (aLab `join` bLab);
   ComponentB.computeSum sharedSecret;
} @ aLab;
```

By using this label in the above bracket, access to `secretNum` is further limited to only components possessing the clearance authority associated with `bP`. Using this design, if a malicious program somehow substituted itself for component B, it would not be able to access `secretNum` (assuming it did not somehow gain access to B's clearance authority as well). As an example, if component B represented a specific DLL and captured B's authorities, a DLL injection attack might be prevented. A is essentially limiting the use of `secretNum` to a component with access to B's authorities.

Sending Data on Channels Conveniently:

Data received from channels in Breeze are always tainted by the channel's label. Regardless of its initial label, data successfully sent on a channel will always be tainted by the channel's label on receive in order to avoid leaking information. If the label is too high for the data to be sent, a NaV will be thrown instead.

Sometimes, it may be desirable to retain the data's original label and avoid this taint. An easy solution to this is to put the data in a "box" before sending it over the channel. This way, the box gets tainted by the channel's label but protects the data's original label on the inside. It is important to note that this does not violate any IFC policies and is simply a convenient technique used in Breeze application development. An example of a boxed channel is shown below:

```
datatype BoxType =
   | Box (Any)
;
```

```
fun unbox {box : BoxTypeC} : Any =
   case box of {
      | Box contents => contents
   };

fun sendBoxed {c : Chan} x =
   send c (Box x);

fun recvBoxed {c : Chan} =
   unbox (recv c);

chan = channel public;
sendBoxed chan 37@top;
test (recvBoxed chan) ==> 37@top;
```

A full implementation of the `BoxingChan` module is available in the release examples directory, and uses sealing to mimic the `Box` behavior. Another way to achieve this behavior is to use "messages", as done in the multi-threaded sum server tutorial to implement requests. The sum server acts on a `SumMessage`, which has the form:

```
datatype SumMessage =
   | SumRequest Int Int (Int ?=> Unit)
   | QuitRequest
;
```

In this case, a `SumRequest` message sent to the sum server protects the labels on the two integer arguments and the response channel.

How and When to Declassify Data:

When declassification is necessary (such as outputting a web page on a web server), it is typically a better practice to wait as long as possible before declassifying. As a simple example of a function that requires a declassified return value, instead of declassifying all the arguments to a computation before execution, it would be beneficial to execute the computation with a raised pc before finally declassifying the result and returning the value. This ensures all functions called are done so with least privilege and mutual suspicion. In the following example, a Boolean `success` will be sent over a public channel to the user to notify whether a login was successful. The example demonstrates a poor use of IFC for protecting sensitive information (in this case, the user's password).

```
// example of a poor placement for declassification
inputPass = "password123" @ serverLab;
inputPassPub = lowerLabelTo serverPc inputPass public;
inputPassHash = PHP.hash `SHA1 passPub;
success = (inputPassHash == dbPassHash);
sendToBrowser success;
```

In the event that the computation being called is malicious, having declassified the arguments prior to the execution would allow the malicious code to export secret data. In the above example, a malicious PHP library could export the user's password, which is given to the library declassified. The following example uses a more appropriate use of declassification to avoid this potential risk.

```
// example of a good placement for declassification
inputPass = “password123” @ serverLab;
successSecret = {
   raiseClrBy serverClr serverLab;
   raisePcBy serverLab;
   inputPassHash = PHP.hash `SHA1 inputPass;
   (inputPassHash == dbPassHash);
} @ serverLab;
success = lowerLabelTo serverPc successSecret public;
sendToBrowser success;
```

By determining the success Boolean within a bracket, IFC restricts the PHP library from exporting the user’s password. In general, declassification should be done sparingly and as late as possible to minimize the surface area of a potential threat.

4.2.1.3 Lessons Learned

Developing secure applications in Breeze provided a great deal of insight into where threats may occur and how to handle them. While it certainly increased the difficulty in developing useful applications, we discovered a number of new mechanisms and patterns to assist secure application development, e.g., brackets, NaVs, and the propose-and-verify pattern.

(For detailed information on the use of the Breeze language, see the Breeze Manual attached to BAE TR-2803 v4.0.)

While explorations of Breeze as an application development language yielded a number of insights on information flow programming techniques, we did not find Breeze to be an effective system-level language. This is partly due to the fact that there were features of Breeze and its runtime that would have posed implementation challenges if written directly in SAFE assembly language (e.g., bi-directional channels, support for programmable top, bottom, and default label values in custom – non-secrecy – label models,). For example, the immutability of values in Breeze does not easily translate to the kinds of operations that are natural in hardware, where generally everything must be mutable. The inability to rely on the immutability of values had a particular impact on the differences between the stream models offered on Breeze and on the hardware. In order to support a Breeze-style stream model on the hardware, we would need to have deep-copy support in the hardware, which would have been difficult to implement and an unusual inclusion in the hardware.

In spite of the co-design efforts to make the hardware support the desired application language as much as possible, some features are best built on top of other layers of abstraction (in this case Tempest).

The more fundamental reason why Breeze did not become a systems language for SAFE was that the SAFE machine was generalized beyond the label models that Breeze supported. The SAFE machine had features for general label models, via its tagging mechanism including some models that were rooted neither in IFC nor in non-interference, such as our signature label model, which omitted several label flow paths.

The Breeze semantics also failed to reflect some the linearity properties of values in the SAFE hardware. This in combination with the other mismatches between Breeze and the SAFE hardware led us to develop a lower-level language called Tempest for systems implementation instead. Once the lower-level language was implemented, we found that it was usable enough for

implementing the demonstration applications that we needed, and so work on Tempest superseded the implementation of the Breeze-to-SAFE compiler.

4.2.2 Tempest

Tempest is a low-level language with record types, structural subtyping, kind-constrained polymorphism, and support for SAFE features such as linear values, brackets, gate construction, tagged static data, and explicit calling conventions. Tempest was inspired by Cyclone (Jim, Morrisett, Grossman, Hicks, Cheney, & Wang, 2002) and Rust (https://www.rust-lang.org/), but takes advantage of the fact that it runs on the SAFE platform with its dynamic checks, allowing for the expression of possibly unsafe operations that are checked for safety by the hardware at runtime. For an introduction to Tempest, details on programming in Tempest, and examples of Tempest programs, see the Tempest Tutorial document attached to the SAFE Computer Programming Manual (TR- 2803).

There are many features of the SAFE system that were exposed to Tempest programs as libraries that were implemented using embedded assembly language blocks. These features are discussed in Section 4.1.1 on the SAFE ISA, but have no special language support from Tempest because we found that they were adequately usable via functions provided by the standard library. Since there is no special language support for these features, they are not discussed below.

Due to the rapidly evolution of Tempest, there is not a full account of the static semantics of the language. We provide here an overview of the novel parts of the static semantics here, including calling conventions and the can-call relation, the kinding discipline, and the focusing for non-destructive use of linear pointers. The other parts of the static semantics are essentially standard, though their precise specification and implementation is complicated by the need to support the novel features mentioned. Tempest also includes features developed in Breeze that are part of the underlying SAFE system, such as brackets and the ability to manipulate tags. These features are also discussed below.

Explicit calling conventions. Tempest requires users to explicitly define a register-based calling convention as part of the type of each function. The calling convention determines

- in which registers a function expects its arguments,
- which registers are available for reading or writing during execution,
- in which registers a function will place its results, and
- which allocators (register-spilling or heap) are available during execution.

For example, the calling convention

```
type UserWareCC () = cconv { 1 2 3 4 5 6 7 8 9 10 -> 1 2 3 4 5 6 7 8 9 10;
                             0 .. 30 : AVAIL
                             31 : ALLOC };
```

is the standard convention to use for application code that does not require the additional efficiency that can be gained by avoiding allocation of space to spill registers. The calling convention `UserWareCC` specifies that registers 1 through 10 are to be used for both input and output, that registers 0 through 30 are available both for reading and for writing, and that register 31 contains the OS services frame, which must at least contain services for register-spilling and heap allocation.

One function may call another whenever the registers of the calling function are at least as available as the registers of the called function. In addition, the called function must require a subset of the OS services that are available to the calling function. Register 0 is exempted from the availability requirements because of its special treatment by the SAFE hardware. Since register 0 is always saved and restored by the hardware on a call and return, it is treated as if it is always available to the called function, even if the calling function's calling convention specifies otherwise.

An alternative to a calling convention is to specify that a function is to always be inlined. This avoids call instructions and stack manipulation at the cost of increased code size, the inability to define the functions recursively, and the inability to use those inlined-only functions as arguments to higher-order functions.

The inclusion of explicit calling conventions arose from the lack of an efficient call stack in the Tempest runtime. In order to fully leverage the pointer bound checks that were supplied by the underlying system, a new stack frame had to be allocated on each call. Explicit calling conventions allowed users to structure their programs to avoid this cost by structuring the code so that spilling was unnecessary and checking this by declaring that functions did not include a service for the allocation of register spill space.

Near the end of the project, we were converging on more uniform calling conventions for Tempest. In particular, restrictions on the ISA imposed by the need to separate the result tag from the available registers during a bracket call may have forced the Tempest compiler to treat some registers as always available for both reading and writing, because of the lack of room in a single instruction to specify otherwise.

Along with this change, we intended to force all calling conventions to use register 31 as the register holding OS services. This change would simplify the compiler and generated code without impacting the efficiency of Tempest programs (which all either require an OS services frame or do not require that many registers). The change also would make it possible to include a greater variety of services in the calling convention declaration without making the specifications unreadable. The ability to specify different services in the calling convention would allow additional OS service dependencies to be statically checked.

We also discovered that users do not like having to specify calling conventions or worry about register usage while writing application code. In particular, standard library code and applications using the standard libraries would have been much easier to design if Tempest had a single stack-based calling convention, with some effort put into designing a more efficient stack allocation strategy that could still leverage the pointer bounds protections offered by the underlying SAFE system.

Kinding. The Tempest type system is designed to reflect the hardware types provided by the underlying SAFE system. In particular, Tempest has a kinding discipline that distinguishes between the *linear* and *unrestricted* parts of variables and expressions. In the following example, values of the `MyPair` type will occupy two registers at runtime. The left component of the type will only be manipulated using instructions from the SAFE ISA that can operate on linear values, even though unrestricted values can be stored in the left side of the pair.

```
newtype MyPair@[s : L, t : U] : L*U = { left : s; right : t }
```

This example also demonstrates the limitations on parameterized types in Tempest. Because values vary in width and kind, types can only be parameterized up to a particular kind. With this in mind, most of the Tempest containers libraries make the assumption that a user will allocate memory for values and only store pointers to the values in the library. This limitation allows us to generate a single instance of the code for each function. The assumption about user behavior allows us to provide general purpose libraries with no more overhead than is incurred in languages that automatically place all values on the heap. Additionally, whenever the type of the values to be stored in a container has kind `U`, the values can be stored "unboxed" without any additional work from the user.

Unlike many systems with linear types, one part of a type being linear does not force the whole type to be treated as linear. This design decision is useful when writing low-level code, but can cause problems in higher-level code, where code that appears to inspect part of a structure can be destructive to that part of the structure, leaving the whole value in an inconsistent state. To mitigate this problem, Tempest includes a rudimentary extension to its use-before-definition checker, so that some uses of destroyed values can be detected. The extension works by marking a variable as uninitialized whenever it is destroyed, so that the usual use-before-definition check can detect problems. This creates some overhead when a conditional that is split into two parts destroys a linear value in only one branch. To avoid the spurious error, the use-before-definition checker requires that a NaV (written `Error` in Tempest) is explicitly assigned to the destroyed value, as in the following:

```
if (b) {
   destructiveOperation(v);
   v := Error;
} else { }
doSomething();
if (!b) { destructiveOperation(v); }
else {}
```

Focusing. Tempest includes a form called *focusing* to make it easier to deal with complex linear data structures. Focusing allows a linear pointer to be offset for the duration of a block and then automatically restored at the conclusion of the block. That is, a pointer `x : LFP(Int[10])` could be focused in the expression

```
focus p = x.(2) in { … }
```

so that within the block `x` would not be in scope and `p` would be in scope with type `LFP(Int)`.

Within the block the ability to use the focused pointer is limited: it cannot be moved to other variables, explicitly stored to memory, passed to functions, or returned from the block. The focused pointer can be read from and written to (i.e. memory may be accessed through the pointer). These limitations ensure that the compiler can safely restore the pointer at the end of the block. Expressing the limitations in the typing judgments for Tempest required the introduction of a second environment for tracking variable scoping.

We suspect that some of the limitations on the use of focused pointers could have been removed by the addition of either kinds that disallowed destructive operations or by introducing "restoring" arguments to functions that would disallow destructive operations. However, the original implementation of Tempest impaired the ability to change the compiler to support those new features.

Brackets. Tempest's brackets are derived from Breeze, but tailored for the limitations of a hardware implementation of a bracket call. In order to support non-interference enforcement by label models, the SAFE ISA `breturn` instruction clears all registers not marked as read-only or dedicated as the return register, tags the return register with the pre-selected tag, and restores the PC tag. The registers to be cleared, the return register, the current PC tag, and the tag for the result of the call must all be selected at the time of the bracket call.

In Tempest the registers to be cleared are those marked as writable in the calling convention. The return register and the fact that a function is to be used via a bracket call are also marked in the calling convention. This information must be part of the function to be called because bracket calls utilize a different return instruction from normal or gate calls, and so the special return instruction must be generated independently of the calling code. A discussion of the ISA design decisions on this topic can be found in Section 4.1.1.

Tag (label) manipulation. Tempest's features for tag manipulation are mostly provided by standard libraries that are implemented via embedded assembly. The exceptions to this are the tagging of data returned from bracket calls, as discussed above, and the ability to tag static data appearing by the use of an `atomtag` expression, which causes all following literals in the same or nested blocks to be tagged with the given tag literal.

It is worth reiterating that in Tempest tags are attached to *values*, not variables. This is because there is no static label model enforcement in Tempest. Therefore, in an expression like

```
var x = {atomtag Private; 42}
```

the tag Private is attached to the value 42 and follows it as the value is passed to functions, stored in memory, or sent to other processes via streams.

Difficulties in Tempest. The presence of linear values limited the kinds of optimizations that could be done by Tempest. In a language with linear types running on a traditional system, copy propagation can be used without violating properties guaranteed by the language because the underlying system does not enforce the linearity of values. Because on the SAFE system *values* are linear, and must be moved instead of copied, this optimization strategy does not work.

Similar problems were presented by the bounds checking during pointer manipulation when trying to do standard optimizations. Typically any code that offsets a pointer and immediately restores it by offsetting the same amount in the other direction would be eliminated. However, on SAFE it is not guaranteed that value would remain a pointer after being offset, because offsetting a linear pointer beyond its bounds results in a NaV, with no ability to recover the original pointer. In this case we were able to perform some optimizations by only eliminating code with this behavior that was generated by the Tempest compiler itself (i.e. not code that was included in an embedded assembly block) and by asserting that there was no guarantee of behavior if a user cast a variable so that the compiler generated assembly would have failed. For example, the following generated code, which offsets a linear pointer by 5 and then offsets the resulting pointer by 5 in the other direction, should be eliminated:

```
offlp t 5
offlp t -5
```

However, without our additional assumptions we would not be able to safely do so.

More insidious than either of these difficulties is the inability to give an operational semantics to Tempest without imposing constraints on the label models that may be present on the system. Some label models are violated by basic book-keeping operations done by a Tempest program. For example, we had to use a signature label model instead of a non-interference integrity label model because of the interactions between that label model and the need to spill registers in Tempest programs.

One way to address the problem would have been to place constraints on what kinds of label models were allowed to be used on the SAFE system for Tempest programs to work correctly. However, the rapid evolution of both the requirements for and the implementation of Tempest and the label models that were in use prevented us from identifying the required properties of an acceptable label model during the project.

4.3 SAFE Operating System (OS) – Concreteware

The operating system of the SAFE machine consists of a number of components that operate with individual authorities to provide a least privilege implementation of the system software. These components are described below:

PAT Server

The PAT server performs all **p**rincipal, **a**uthority, and **t**ag related operations. It is the only part of the system that can operate on the internals of the tag or authority representation. Currently, principals are represented as atomic tags.

If a thread takes a TMU miss, the PAT Server determines the correct result by examining the M vector. The PAT server works with a product label model—a label model made up of more than one label model. In order to execute an instruction, all label models must allow it. The resulting tags from each label model are then combined into the final Tags that are returned.

The PAT server can also create a new tag and authority upon request and return them to the calling thread.

Because the PAT works with multiple label models, it also provides an API for threads to request a tag that only contains a particular label model. This is important for other operations.

The PAT server also exposes an API to allow threads to request the result of a meet or join operation, as well as whether or not one tag can flow to another, or if they are equal.

Lastly, the PAT server exposes an API to allow threads to retrieve the Default tag and the Bottom tag. The default and bottom tags are not necessarily identical, though they are in the secrecy label model. The default tag is made up of what all label models consider their default value. If a tag doesn't include any information specific to a label model, it is considered to be the label model's default value. The Bottom tag is the tag comprised of the tags representing the bottom of each label model's lattice.

Many label models were written, but not all are in use. Currently implemented and in use are a secrecy label model and a signature label model. The secrecy label model prevents secret data from being leaked, while the signature label model is used to ensure data is not tampered with. All label models must be representable as a lattice, but otherwise there are few limitations on the label models that can be implemented. Label models are described in general in Section 3.1.

Instructions for adding new label models to the PAT server can be found in appendix C of the Computer Programming Manual (TR-2803).

Label models installed for use can be changed at compile time, but not at runtime. Once the system is running with a set of label models, they remain active.

Scheduler

The scheduler currently implemented in SAFE is a pre-emptive round robin scheduler. This means that each thread is allowed to run for a maximum set length of time, after which it is interrupted and the next thread is allowed to run. Threads are run one by one, always in the same order. After all threads have been run, the first thread is run again.

When running a thread, the scheduler first checks if the thread is in a runnable state. If so, the thread is run as normal. If not, the scheduler checks the thread state to determine the cause of the error. It then runs the appropriate fault handler. Once the fault handler has finished, it switches back to the main thread, which continues running as if the error had never occurred.

Threads are, of course, allowed to yield their remaining time if they cannot make further progress. Some stream instructions will yield if they cannot currently complete. This yielded time is lost – the next thread to run does not benefit from extra run time.

Stream manager

Because SAFE is designed to avoid using shared memory, threads can only communicate with each other via streams. Streams are one way communication. One thread can write to the stream while another thread can only read from the stream. User threads can only communicate with the concreteware (CW) or other threads via streams, though in the case of CW, the stream calls are concealed within accessor functions.

Memory manager

SAFE has both global and per-thread allocators. The global allocator is implemented as a combination of a buddy allocator and a static allocator. The buddy allocator divides up the large chunk of memory it manages into smaller and smaller chunks until it reaches the minimum size that will hold the requested memory. The static allocator has many frames of sizes up to 64 atoms pre-allocated at compile time. It then can simply hand out one of the pre-allocated frames. The buddy allocator is faster for large allocations, so the two allocators are combined to improve performance.

The global allocator is important for spawning a new thread or allocating new tags and authorities. Due to the overhead in calls to the global allocator, multiple requests can be sent at once.

There are two thread-local allocators: a heap allocator and a stack allocator. The heap allocator is used for normal allocations, while the stack allocator is used for register spilling.

The local heap allocator is implemented as a bump allocator. Unlike the global allocator, which operates on a large amount of memory that can be used by any thread, the local allocator allocates from a thread-specific region of memory that other threads do not have any access to. In our design, pointers to global memory can be sent across streams to other threads (e.g., pointers to streams), while pointers to thread-local memory cannot. Instead the system should

make a deep copy of the pointer in the receiving thread's memory and sends the copy. Due to the difficulty of the interaction between the secrecy label model and the procedure for creating a deep copy, this functionality was not completed. See the section on garbage collection for details about these difficulties.

Threads can make use of the global allocator only via communication to a Concreteware service such as the stream manager, which requires global allocation so that threads can communicate via the shared memory.

The allocator for register spilling is a simple stack allocator. The Tempest compiler calls the stack allocator (if necessary) at the start of each function call and explicitly frees the memory at the end of each function call, or before a tail call.

Thread manager

Each thread in the SAFE system is made up of numerous hardware threads: the main thread and all of its fault handlers. Each thread also starts with pointers to the services (e.g., allocator, PAT server) it has available.

To spawn a thread, the creating thread calls the thread manager. The thread manager allocates the memory for the thread and all of its fault handlers. The thread manager also sets the services register and provides two streams for the parent and child thread to communicate and a third stream for the parent thread to receive a notification if child thread crashes.

Once the thread has been created, the thread manager adds the new thread to the scheduler.

Tempest standard libraries

Most "primitive" operations in Tempest, such as arithmetic and Boolean operators, are provided by standard libraries that wrap inline assembly code with functions. There are libraries written in Tempest for various standard data structures, such as linked lists, array lists, stacks, tuples, and strings. There are also libraries for printing and logging, which allow for strings to be written to streams that are redirected to standard-output on the host machine for the safe-sim and SAFE FPGA implementations.

The generation of values for string literals in Tempest makes assumptions about the concrete representation of strings in the string library, tightly coupling the standard string library and the Tempest compiler implementation. The same is true for the library implementation of the gates that are used to access OS services.

Device Drivers

Currently the SAFE machine has no device drivers to handle a mouse or keyboard, however this does not prevent interactive programs from being run. All interactivity in programs is handled through streams that read and write information to and from outside of the SAFE machine. This information required to be tagged Public to prevent leaking information.

The system also has a Tempest library that allows communication through the UART on the FPGA. The UART is accessed using memory mapped I/O.

4.4 Demonstrations

During the course of the SAFE project we deployed 3 progressively more sophisticated demonstrations: “sum server”, SAFE Knowledge Online (SKO), and the Rocket Controller.

The **sum server** demo is a command line demonstration of the overall SAFE architecture in the context of a simple information flow control example. “Sum” refers to the use of a simple arithmetic example to add data with different labels. Multiple computations are performed in the demonstration. The result of the first is a TMU cache **miss** since the label combination presented has not been seen before. For example adding 1@{SS,A,B} + 2@{SS,A} yields 3@{SS,A} (the symbols in the {} are data labels). The second computation uses a set of labels that has already been seen and results in a TMU cache **hit**. The final example uses data that cannot be read by the executing program and results in an access error violation and a NaV value. The purpose of this demonstration was simply to confirm operation of the basic machine mechanisms.

The other two demonstrations are described in the following sections.

4.4.1 SAFE Knowledge Online (SKO)

The SAFE Knowledge Online (SKO) demo is an information flow control demo that shows privacy protection of data similar to that found in Army Knowledge Online (AKO). Member data includes: username, name, phone, email, rank, organization, account type, year joined, years of service and affinity group. Fields in the database were labeled as either public (e.g., name), private to the user (e.g., phone and email) or private to the affinity group (e.g., rank and year joined).

The demonstration shows results of database queries for different kinds of users. If you are not logged in then any data more secret than public is redacted (blocked from being returned by the SAFE machine). If you are logged in as a particular user then you can see your own phone and email (but no one else’s) and you can see the username, rank, and year joined for other members of the same affinity group.

The components of the demonstration include a SAFE machine running the garbage collector, the PAT server, scheduler, and TMU handler. An application thread runs on the machine that serves up a web-based demonstration by: parsing http requests, authenticating the user, adding authority to the computation, handling the request, declassifying data to be displayed, and constructing an http response. Networking for the demonstration is handled by a front-end Linux-proxy box which runs the TCP/IP network stack and handles the conversion of data from SAFE atoms to (untagged) byte streams.

The database application is written in Tempest. Data is explicitly tagged in the database application using built-in language labeling features and an application defined secrecy label model formed from individual and conjoined principal read authorities.

The value proposition of this demo is to show that independent of the complexity of the application, SAFE can provide an information-flow security perimeter to prevent exfiltration of sensitive data. After this demonstration, the Home Depot and Target penetrations occurred. Credit card data labeled as secret would not be able to escape from a SAFE-based machine.

4.4.2 Rocket Controller

The SAFE Rocket Controller application demonstrated how the protection mechanisms of SAFE prevent various classes of attacks even in the face of poorly or maliciously written application code. The demonstration includes protections against:

- code injection via buffer overflow,
- control flow hijacking via jumping according to user input,
- SQL injection to create false data, and
- password leaking via unauthorized access to an administrative database user interface.

The Rocket Controller application was designed to allow all of the attacks to be attempted, so that the SAFE protection mechanisms would have to come into play.

The normal use case for the application is:

1. A targeter proposes a target, authenticating to the system with a username and password.
2. A commander approves a target, authenticating to the system with a different username and password.
3. The targeter selects an approved target and fires the rocket, again authenticating to the system.

In addition to this, there is an "administrator" interface that allows for the querying of arbitrary database tables, including the password table. The Rocket Controller backend ran as an HTTP server on the SAFE system. The user interface was provided by a web page hosted on the host machine. Access to the SAFE system was done via AJAX requests from the web page that were redirected from the local webserver to the SAFE system.

There are three places in the system where we ensured a correct, non-malicious implementation: the passwords in the table were tagged secret, so that they were only available to the password manager, the signatures of the targeter and commander were only given upon correct successful authentication, and the firing operation checked for the signature of the commander before actually firing the rocket. These are exactly the operations that we would expect to be subject to audit in a SAFE system. With these three operations protected, the failure of the rest of the system can be successfully protected by the SAFE protection mechanisms.

See Figure 3 for a depiction of how requests to the Rocket Controller were handled.

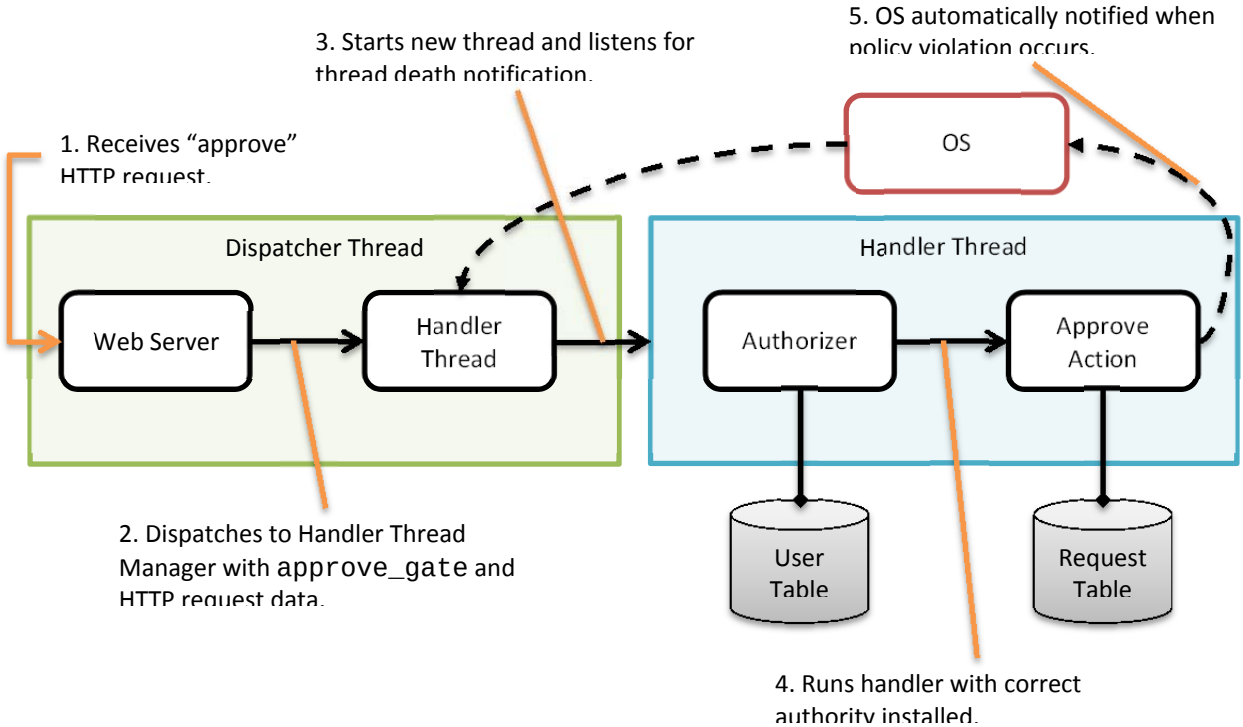

Figure 3: Rocket Controller Control Flow

Code injection. For the code injection attack, we intentionally omitted a bounds check on user input and instead naïvely copied the username for authenticating to the Rocket Controller application. By making the user input longer than the buffer available to hold the name, the user was able to attempt to overwrite data or code in the system. However, the bounded pointers on the SAFE system protected against this attack, and instead the thread that was copying the data crashed, resulting in an error for the user.

Control flow hijacking. For the control flow hijacking attack, we stored a gate pointer for processing data in the same frame as the data to be processed. This allowed the attacker to actually make use of the buffer overflow to overwrite the pointer that would then be called. However, since the atomic group of the data written over the gate pointer was an integer, the attempt to jump to the pointer failed with an atomic group error, resulting in an error for the user.

SQL injection. For the SQL injection attack we used naïve string concatenation to produce the SQL-like string that was interpreted on the SAFE system to do database access. This allowed a malicious targeter to inject extra SQL code to set the target approval field for an unapproved target. However, since the signature of the commander was unavailable to the targeter, the check for the signature of the commander when the rocket was fired failed.

Unauthorized database access. The administrative database access allowed for arbitrary queries of the data. However, all data leaving the SAFE system that was not tagged as public was replaced with a default error string. Even though a malicious targeter was able to query the password database using the interface, since there is no way of logging in as the password

manager, there is no way for the passwords themselves to leave the system. Thus, the query resulted in a useless response for the attacker.

In contemporary attacks like Stuxnet there are cascading series of failures which ultimately allow the attacks to succeed. With the Rocket Controller demonstration, we demonstrate SAFE providing a cascading series of successful protection.

4.5 SAFE Tools

SAFE project development efforts spanned a significant set of interrelated tasks. The SAFE platform was designed and built as a clean-slate effort, applications and demonstrations were built to illustrate the capabilities of the platform, and, as described in this section, a significant number of tools were built to support the development and debugging of the SAFE platform and applications. In the conclusion of this report we describe lessons learned in juggling the development of these three complementary capability branches.

4.5.1 SAFE Machine Simulator: safe-sim

The SAFE software simulator, or safe-sim, is a simulator that can run programs written in the SAFE assembly language. It is written in Haskell. While slower than the FPGA implementation of SAFE, safe-sim provides additional debugging capabilities, and allows for developing programs even without immediate access to the FPGA implementation. The software simulator attempts to be as faithful to the hardware implementation as possible, with one notable exception.

The TMU cache is implemented differently in safe-sim than in the actual hardware implementation. The hardware has a limited cache space, while safe-sim is free to keep as many rules cached as memory allows. Therefore, safe-sim does not make use of the same hash functions that the FPGA simulator uses to keep track of tag rules. This is visible on occasion when the hardware takes a TMU miss on a rule that it has previously resolved, but that has been forced out of the cache. safe-sim will not take a TMU miss on the same instruction, as it still has the rule cached.

The simulator can be run in two modes: interactive and non-interactive. In non-interactive mode, the simulator loads the program into memory and immediately begins execution. When the program has halted, the simulator exits. It reports the final PC and the number of instructions run. If the machine did not halt normally, the simulator will print an error message. In interactive mode, the simulator loads the program into memory, and then prompts the user for input. The user can chose to set breakpoints or watchpoints, modify memory, change the amount of information the simulator prints as it runs, or run the program, either indefinitely or for a specified number of instructions.

Watchpoints are a debugging feature added to safe-sim. A watchpoint can be set to trigger on the read and/or write of a memory address. If the watchpoint is triggered, the simulator will stop running, print a message, and ask the user for input again. In the case of a watchpoint on memory write, the simulator will print both the original value of memory and what it was changed to.

Another debugging feature adding to safe-sim was the ability to save a machine state in order to resume from it at a later point. This allowed the person debugging a program to mark when the program was in a known good state while they attempted to locate where it went wrong. This helped reduce the time wasted if the user finds when they reach where they think the error will

occur only to find that something already looks wrong. At that point, they could restore to the earlier state and set a more informed breakpoint.

There are some error conditions where the hardware cannot continue, but due to limitations also cannot report details about the error. The safe-sim simulator attempts to address many of these conditions by printing error messages in the case of un-recoverable errors. For example, if the initial boot thread is corrupted, the hardware cannot output an error message. The software simulator can print the error before exiting.

The simulator's usage is documented in the safe-sim User's Guide, which is an appendix to the Computer Operating Manual (BAE TR-2748).

4.5.2 SAFE Assembler

The SAFE assembler, or safe-asm, is an assembler for the SAFE assembly language. It is written in Haskell. From an assembly file, safe-asm can produce a binary image file for use with the FPGA or bsim (the Bluespec simulator of the hardware implementation). Rather than output a format for safe-sim to read, safe-sim calls the assembler as a library. This gives safe-sim more information to assist the user in debugging. When assembling an image, the assembler can also produce a "map" file that shows the final memory addresses and some information about their source.

The assembler can also output a configuration file for the FPGA implementation, as well as many other things that may help someone writing pure assembly code. This configuration file contains constants such as the memory size that must be consistent in all simulators. This allows for some variables to be changed without needing to modify the Bluespec code defining the hardware implementation, though the Bluespec code will still need to be recompiled.

While most of the configuration information the assembler uses is specified in the ISA specification, some details are defined only by the config file itself. This includes the encoding of the atomic groups and the hash functions used in the Bluespec code for the TMU hashes.

The assembler also contains a disassembler that can convert the binary atoms back to their internal representation. This is used as a library in safe-debugger and safe-sim.

4.5.3 SAFE Linker: safe-meld

The design of safe-meld is based on "Units: Cool Modules for HOT Languages" (Flatt & Felleisen, 1998). A meld linking specification file is an abstract specification of the linking result and a declarative specification of a hierarchical (nested) collection of units that consist of a collection of SAFE assembly files with specified imported and exported names. Available linking results were object files, whole-system images, and loader-formatted images.

Constraints around data loading due to the need for loader privileged retagging of instruction description data as code lead to the use of whole-image linking. Linking specification files with generative behavior for individual components meant code sharing was optional, which was important for certain libraries that utilized some amount of "global" mutable state.

The linker itself consisted of three parts: meld-core, which creates linking plan (consisting of renaming and hiding), safe-meld-lib, which provides a compilation unit representation and the parser for the specification language, and safe-meld which manages the linking process. The concrete compilation unit representation as produced by the Tempest compiler did not include

meta-data to support linking (as most object files do in other languages), which slowed down linking but made it easy to avoid bugs. The slowdown was only a problem near the end of the project, because of the need to link the whole OS for each program.

The safe-meld component also had some built-in compilation units to make it possible to specify streams, data frames and hardware thread frames statically in linking specification files. This feature was especially useful when declaring whole-system images, since it allowed for the concise declaration of large data blocks that were used by the SAFE OS services.

Usage instructions for the safe-meld tool are provided as part of the Tempest Tutorial document attached to the SAFE Computer Programming Manual (TR- 2803).

4.5.4 SAFE Debugger: safe-debugger

The SAFE debugger is a Haskell program to assist in running and debugging programs on the FPGA or in bsim. When running non-interactively, the SAFE debugger will time out after a period of time even if the machine does not halt. Since the FPGA executes programs quickly, this allows the SAFE debugger to avoid waiting for a program that has entered an infinite loop to finish. The time out is adjustable for longer running programs.

It provides an interactive interface, allowing for the user to set breakpoints and single step, as well as to examine and modify memory. safe-debugger provides much of the same functionality as safe-sim.

4.6 Verification Efforts

The design of the SAFE processor was influenced by verification considerations and efforts to verify critical aspects of the design. A lot of effort was spent on designing languages that provided a non-interference property. The verification effort was orthogonal to the primary platform design effort in the sense that verification was performed on models of the components as opposed to the actual component implementations. The verification efforts are well documented in the published literature:

B. Montagu, B. C. Pierce, and R. Pollack, "A theory of information-flow labels," in *26th IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 2013, pp. 3–17. Available: http://www.crash-safe.org/node/25

> **Summary**: SAFE's tags are intended to support a wide range of dynamic analyses, including both information-flow policies and others such as memory-safety and control-flow-integrity policies. This paper focuses on the first, offering the first generic characterization of information flow in the setting of a simple functional programming language. A number of existing information-flow policies are shown to fit within this framework.

C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett, "All your IFCException are belong to us," in *34th IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2013, pp. 3–17. Available: http://www.crash-safe.org/node/23

> **Summary**: Dealing correctly with exception handling in the context of

information-flow tracking turned out to be one of the most challenging issues in designing a high-level programming language to run on the SAFE processor. This paper examines the topic in detail, proposes a specific mechanism, and proves it correct.

C. Hriţcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos, "Testing noninterference, quickly," in *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Sep. 2013. Available: http://www.crash-safe.org/node/24

Summary: This paper attacked the issue of how to design and debug an information-flow policy for the SAFE machine. Such policies are quite tricky to get right, and tend to be subject to quite subtle security bugs. This paper investigated the extent to which property-based random testing techniques could be used to speed the design process by identifying many bugs early in the process.

A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach, "A verified information-flow architecture," in *Proceedings of the 41st Symposium on Principles of Programming Languages*, ser. POPL. ACM, Jan. 2014, pp. 165–178. Available: http://www.crash-safe.org/node/29

Summary: This paper brings together many of the earlier results from the theory and verification side of the SAFE project, showing how to specify and formally verify an information-flow policy implemented by tag-propagation rules on an idealized microprocessor enhanced with a simple version of SAFE's rule cache and tag propagation mechanisms.

A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach, "Micro- policies: Formally verified, tag-based security monitors," in *36th IEEE Symposium on Security and Privacy (Oakland S&P)*. IEEE Computer Society, May 2015, pp. 813–830. Available: http://prosecco.gforge.inria.fr/personal/hritcu/publications/micro-policies.pdf

Summary: This follow-up to the 2013 POPL paper showed how to generalize the specification and verification framework presented there to a wide range of micro-policies including dynamic sealing, memory safety, control-flow integrity, and software compartmentalization.

4.7 Hardware Design and Optimization

The design of the SAFE hardware is well-documented in the published literature. While efficient hardware design is crucial to the overall viability of the SAFE secure processor, the hardware provides a well-defined interface to support the operating system, tools, and application software. The following papers provide detailed information about the hardware design.

[DD13] Udit Dhawan and André DeHon. Area-efficient near-associative memories on FPGAs.

In *Inter- national Symposium on Field-Programmable Gate Arrays, (FPGA2013)*, February 2013.

> **Summary**: Describes the efficient hardware structure for performing wide (nearly) associative matches that makes the TMU (Tag Management Unit) or PUMP (Programmable Unit for Metadata Processing) possible. It is described here in generic terms to avoid needing to setup the whole background of the SAFE project and since the unit has broad utility beyond TMU/PUMP use.

[DD15] Udit Dhawan and André DeHon. Area-efficient near-associative memories on FPGAs. *Transactions on Reconfigurable Technology and Systems*, 7(4):3:1–3:22, January 2015.

> **Summary**: Describes the efficient hardware structure for performing wide (nearly) associative matches that makes the TMU (Tag Management Unit) or PUMP (Programmable Unit for Metadata Processing) possible. It is described here in generic terms to avoid needing to setup the whole background of the SAFE project and since the unit has broad utility beyond TMU/PUMP use. This includes appendices with probabilistic analysis of the behavior of the structure and algorithms.

[Dha13] Udit Dhawan. Dynamic multi-hash cache architecture bluespec source distribution. http://ic. ese.upenn.edu/distributions/dmhc_fpga2013/, February 2013.

> **Summary**: Bluespec System Verilog source code for the dMHC. This provides the full code associated with the FPGA2013 (and TRETS2015) dMHC articles. The code is highly parameterized so that it can be easily adapted to various uses.

[DHR+15] Udit Dhawan, Cătălin Hriţcu, Rafi Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and André DeHon. Architectural support for software-defined metadata processing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 487–502, 2015.

> **Summary**: The architecture where we extract the TMU (Tag Management Unit) developed for SAFE and integrate it into a conventional processor and show that it can be used to support safety and security policies on unmodified C-code. This includes quantification of area, energy, and delay costs as well as microarchitectural implementation techniques beyond the base TMU to reduce these costs.

[DK12] Udit Dhawan and Albert Kwon. SAFE processor source distribution. http://ic.ese.upenn. edu/distributions/safe_processor/, October 2012.

> **Summary**: Bluespec System Verilog source code for the snapshot of the SAFE Processor associated with the AHNS 2012 Interlocks paper. This is the full processor core at that point. This version was not fully pipelined, but operated in phases. This release is only the processor Bluespec, so lacks the rich set of tools

for generating code that works with the processor.

[DKK+11] André DeHon, Ben Karel, Thomas F. Knight, Jr., Gregory Malecha, Benoît Montagu, Robin Morisset, Greg Morrisett, Benjamin C. Pierce, Randy Pollack, Sumit Ray, Olin Shivers, Jonathan M. Smith, and Gregory Sullivan. Preliminary design of the SAFE platform. In *6th Workshop on Programming Languages and Operating Systems*, PLOS, October 2011.

Summary: Describes entire stack including hardware, software, and programming language.

[DKK+12] Udit Dhawan, Albert Kwon, Edin Kadric, Cătălin Hriţcu, Benjamin C. Pierce, Jonathan M. Smith, André DeHon, Gregory Malecha, Greg Morrisett, Thomas F. Knight, Jr., Andrew Sutherland, Tom Hawkins, Amanda Zyxnfryx, David Wittenberg, Sumit Ray, and Greg Sullivan. Hardware support for safety interlocks and introspection. In *SASO Workshop on Adaptive Host and Network Security*, September 2012.

Summary: Describes the SAFE processor. This focused on the key hardware protection mechanisms and included highlights of the FPGA implementation, including the area and timing for the design.

[DVR+14] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Benjamin C. Pierce, and André DeHon. PUMP – A Programmable Unit for Metadata Processing. In *Proceedings of the 3rd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '14, pages 8:1–8:8, New York, NY, USA, June 2014. ACM.

Summary: Preliminary description of the architecture where we extract the TMU (Tag Management Unit) developed for SAFE and integrate it into a conventional processor and show that it can be used to support safety and security policies on unmodified C-code. This workshop paper is largely superceded by the subsequent ASPLOS 2015 paper on the PUMP. The ASPLOS paper was tight for space, so some descriptions are less terse in this paper.

[HDV+14] Cătălin Hriţcu, Udit Dhawan, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Benjamin C. Pierce, and André DeHon. Programming the PUMP: Hardware-assisted micro-policies for security. Unpublished, May 2014.

Summary: This is a good companion to the ASPLOS 2015 paper, describing what the policies are, how they are written, and capturing key characteristics of the policies. The CCS reviewers in 2014 really wanted to see the microarchitectural and optimized implementation details that were in the ASPLOS paper before considering this paper for publication. Nonetheless, we

recommend this material be read along with the ASPLOS 2015 paper to get the complete story

[KD13] Albert Kwon and Udit Dhawan. Low-fat pointers bluespec source distribution. http://ic.ese. upenn.edu/distributions/fatptr_ccs2013/, May 2013.

Summary: Bluespec System Verilog source code for the Low-Fat Pointer base-and-bounds units. This provides the full code associated with the CCS 2013 article. The distribution includes both the optimized BIMA scheme and the original ARIES scheme and is parameterized on field lengths.

[KDS+13] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, Jr., and André DeHon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 721–732. ACM, 2013.

Summary: Describes the hardware scheme for base-and-bounds that provides memory safety in the SAFE processor. This includes an analysis of the area and timing of the base and bounds checks, supporting the particular encoding scheme we chose and quantifying its advantage compared to the initial encoding from Aries. This also shows how to integrate the unit into a pipelined processor and avoid stalls.

[SCD+13] Gregory T. Sullivan, Silviu Chiricescu, André DeHon, Delphine Demange, Suraj Iyer, Aleksey Kliger, Greg Morrisett, Benjamin C. Pierce, Howard Reubenstein, Jonathan M. Smith, Arun Thomas, Jesse Tov, Christopher M. White, and David Wittenberg. SAFE: A clean-slate architecture for secure systems. In *Proceedings of the IEEE International Conference on Technologies for Homeland Security*, pages 570–576, November 2013.

Summary: A short highlight of the entire hardware/software stack for SAFE including verification. This captures a more mature snapshot of the stack than the PLOS 2011 paper, but is necessarily less detailed than the dedicated papers on each of the components. While short, this is the only public publication that describes the operating system design for SAFE

5.0 CONCLUSIONS

Inherently secure processing models are within technological reach today, however, they are not easily compatible with today's existing (legacy) codebases. Based on our experience with a 4 year co-design of the SAFE secure processor we offer these conclusions and lessons learned (summarized in the following list and elaborated in more detail in the remainder of the section):

- Building an entire hardware/software stack in parallel is a Gordian knot which requires clever bootstrapping techniques to resolve.
- Programmability of secure applications is challenging and needs to be an equal partner with secure architecture and language design.
- Providing mechanisms to implement security policies is different than good methods for stating security policies.
- Non-interference is too strict a model to govern useful applications, research is needed in more permissive models.
- Bounded memory frames provide memory safety and support a secure compartmentalized programming model.
- Verifying design models is not the same as verifying implementations.
- Debugging features can leave a big security hole.
- SAFE addressed single host secure processing issues. Many interesting problems remain to support networked, multi-machine secure processing.

Difficulties in building a complete hardware/software stack: For co-design and co-implementation between language and hardware to be successful, the staff working on the hardware need to understand the goals of the staff working on the language and the staff working on the language need to understand the limitations of implementing things in hardware. Otherwise, one of the sub-teams will fix on a design or implementation too soon and lose the ability to respond to the needs of the other. On the other hand, if flexibility is retained too long then the programming language group will be waiting for hardware to firm up and the hardware group will wait for the programming language group requirements. We went through versions of both of these pathologies.

An effective solution is to make use of easily changed simulations to keep the hardware fluid as the programming languages evolves. Unfortunately, the hardware team is not fully-utilized under this model – needing to support the simulation but unable to productively develop and optimize the hardware. Note, that the same issue exists between the programming languages efforts and the operating system and applications efforts. In this case, a co-design approach between applications and the underlying machine effort would be effective with the applications (developed using new secure programming techniques) being used to define platform requirements. However, this approach would have easily extended the project beyond its 4 year boundary.

Secure applications programming challenge: As a project taking a clean-slate approach to the overall program it was unlikely that we could achieve a complete end-to-end lifecycle, application-to-hardware solution. Other projects in the CRASH program built on existing technology and thus were not subject to the Gordian knot discussed above. The focus of SAFE was to explore mechanisms that could provide inherently safe computing and in this goal we were successful. We developed some principles of secure programming but did not deliver a

complete programming methodology for secure systems development on top of this new (and evolving platform). The definition of secure is also something that was explored under the SAFE project and we set a very high bar for security (see non-interference discussed below) and did not distinguish between theoretical security vulnerabilities and readily exploitable security vulnerabilities. This was something of an opportunity missed since we did develop a solution that provided security against whole classes of vulnerabilities, buffer overflow and code injection in particular, and deploying a platform secured against just these exploits would be a valuable contribution in itself (and one that we are working on under a continuing internal effort). However, for such a platform to be a viable solution (or transitionable technology) it needs to be accompanied by a complete software lifecycle methodology defining how to compile, link, deploy, execute, and test the applications (something which the non-clean-slate solutions could obtain from their underling legacy platform).

Stating security policies versus implementing security policies: Current day instruction sets and compilers provide solutions to implementing a variety of computational models. However, instruction set support is not enough. For example, programmers call methods and subroutines without much thought to the underlying complexity of those mechanisms. Use of such features would not be practical if programmers had to explicitly write the code to implement calling and return conventions and stack management.

On the SAFE machine we were able to provide low-level support for security policies implemented as label models (e.g., secrecy/confidentiality enforcement). However, developing code using a low-level non-interference policy is tedious and difficult. More work is required on how to bridge the gap between the application level specification of desired confidentiality properties and the level of enforcement of those properties in the SAFE system. This points again to the need to develop and mature the application level programming paradigm before solidifying the lower-level support capabilities. Whole program security policies involving individually identified principals, symbolic groups of principals, and application roles is an area that requires additional research.

Strict non-interference: A machine is non-interfering if given any sequence of low inputs (the machine has a set of inputs where some are “low” security and other are “high” security), the output of the machine for that particular sequence is the same regardless of what the high inputs are. This means the machine does not leak any secured (high) information. Strict non-interference is a very high bar. For example, imagine a machine that processes both low records and high records and maintains separation of that processing. If that machine happens to count the number of low records and the number of high records as it reads them (the context in which the counting is done is crucial) and keeps a count of the number of high records and outputs that number, e.g., “processed 100 records, 42 high” then that machine has leaked high information and is not non-interfering. While leaking the number of high records could be of concern in some situations, it is a very low-bandwidth leak and does not identify or leak any of the constituent high data.

Any machine that works with high data (non-public data of any sort) is highly likely to need to divulge some data tainted by high data. In the SAFE machine we provided a declassification mechanism as a way to escape strict non-interference.

Granularity of non-interference also matters. Enforcing label models at the level of individual instructions is difficult because the properties we care about are expressed globally. Non-interference properties, for example, are properties of the inputs and outputs of an entire program. The translation of these global properties to instruction-level properties is not straightforward. In the SAFE system, we took the approach of enforcing non-interference for the inputs and outputs of each instruction. This decision meant that there are globally non-interfering programs that cannot be run on the SAFE system because they incur *local* violations of non-interference. For example, we believe it is impossible to implement a garbage collector in such a system without giving the garbage collector special authority to violate the label model.

More research needs to be done to develop information leak properties that are less Boolean in nature and reflect both the quanta of leakage and the bandwidth of any leakage. In the SAFE machine we had to develop programming models that maintained non-interference on an instruction-by-instruction basis which made programming quite difficult.

Bounded memory frames: It is now feasible, as demonstrated in the SAFE platform, with current hardware technology to implement self-identifying bounded memory regions, i.e., memory words that encode the base and bounds of the frame in which they are contained. This single feature provides the basis for memory safety and implementation of a secure compartmentalized programming model. Providing this feature creates a machine that is half-way freed from the current "raw seething bits" design of most contemporary processors (the other half is provided by hardware level data typing that allows, e.g., distinguishing data from instructions from pointers). Given bounded memory, there are many design patterns to be explored that make effective use of the capability (which can be thought of as "coloring" the memory and enforcing policies that maintain the integrity of the colors being accessed). The programming model using bounded memory regions requires additional exploration as much contemporary buffer processing code ignores defined regions to enable more efficient data exchange. Additionally, new patterns need to be developed as the segmentation provided by different memory patterns provides significantly different security protection (e.g., an array of 3 items versus 3 separated independent variables).

Verifying designs: Claims of secure processing ultimately require formal verification. In the literature of published secure mechanisms, there are too many corner cases and examples that have been found with holes. The current state of formal verification provides for verification of models, i.e., designs of computational mechanisms and algorithms (e.g., using the Coq theorem prover). Models, however, are not implemented code and there are few examples of verification technology that can even compile models into executable code and in the case of SAFE, no technology (not surprisingly) to compile models into our unique programming languages and ISA. Attempts at formal verification are nevertheless instructive particularly as verification properties are formulated and weakened or strengthened depending on progress of the verification efforts. This exercise points to issues that must be addressed during the design phase of the secure algorithms.

Securely Debugging: The nature of debugging software is fundamentally one of revealing data. We added certain unsafe features to our SAFE development platform with the knowledge these could not be included in the deployed platform. The current SAFE platform does not provide a debugging solution for the deployed platform. One intermediate solution is to encrypt all debugging output, thus shifting the issue to a key management problem in order to access the

debugging data. Addressing this issue is part of the overall effort that would be required to address the full software creation, deployment, execution, and maintenance lifecycle on a secure host solution.

Networked Secure Processing: While SAFE addresses many secure host processing issues, it was out of scope for both the project and the program to address secure computing across networked hosts. Two primary issues are involved with multi-host processing: persistent data and network data transmission. Data persistence of labeled data requires storage of data using encryption since viewing of the payload data separate from its tag would reveal the data no matter how it was labeled. This results in a key management problem as to how multiple hosts can share the same key(s) for persisted data. Networked data transmission is a similar problem as the data is being serialized over the network (instead of into a file system). Secure communications between hosts can be based on existing secure communications protocols, but they need another level of authentication (or encryption) to allow hosts to prove they are part of a trusted multi-host secure community. In addition to encryption keys, this community must share user-ids (principals) and authorities so that data access restrictions created on one machine can be enforced (with the same semantics as defined on the originating machine(s)) on a partner network machine. Existing secure communications protocols may be adapted to handle this problem, but will also provide a single point of failure should a machine be able to obtain improper access to the secure network.

6.0 BIBLIOGRAPHY

Azevedo de Amorim, A., Dénès, M., Giannarakis, N., Hriţcu, C., Pierce, B. C., Spector-Zabusky, A., et al. (2015). Micro-Policies: Formally Verified, Tag-Based Security Monitors. *IEEE Symposium on Security and Privacy.* Oakland, CA: IEEE.

Chiricescu, S., DeHon, A., Demange, D., Iyer, S., Kliger, A., Morrisett, G., et al. (2013). SAFE: A Clean-Slate Architecture for Secure Systems. *IEEE International Conference on Technologies for Homeland Security (HST).* IEEE.

DeHon, A., Dhawan, U., & Strnad, A. (2014). *SAFE Instruction Set Architecture.*

Dhawan. (2014). PUMP: A Programmable Unit for Metadata Processing.

Dhawan, U., & DeHon, A. (2013). Area-Efficient Near-Associative Memories on FPGAs. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* ACM.

Flatt, M., & Felleisen, M. (1998). Units: Cool Modules for HOT Languages. *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (pp. 236-248). Montreal, Quebec, Canada: ACM.

Hriţcu, C., Greenberg, M., Karel, B., Pierce, B. C., & Morrisett, G. (2013). All your IFCException are belong to us. *IEEE Symposium on Security and Privacy (SP)* (pp. 3-17). San Fransisco, CA: IEEE.

Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., & Wang, Y. (2002). Cyclone: A Safe Dialect of C. *USENIX Annual Technical Conference, General Track* (pp. 275-288). USENIX Association.

Kwon, A., Dhawan, U., Smith, J. M., & Knight, Jr., T. F. (2013). Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for. *ACM SIGSAC Conference on Computer and Communications Security (CCS)* (pp. 721-732). ACM.

Montagu, B., Pierce, B. C., & Pollack, R. (2013). A theory of information-flow labels. *IEEE Computer Security Foundations Symposium* (pp. 3-17). New Orleans, LA: IEEE.

Stefan, D., Russo, A., Mazières, D., & Mitchell, J. C. (2012). Disjunction Category Labels. *Proceedings of the 16th Nordic Conference on Information Security Technology for Applications* (pp. 223-239). Tallinn, Estonia: Springer-Verlag.

LIST OF ACRONYMS, ABBREVIATIONS, AND SYMBOLS

ACRONYM	DESCRIPTION
AG	Atomic Group
AGU	Atomic Group Unit
AKO	Army Knowledge Online
API	Application Program Interface
BROP	Blind Return-Oriented Programming
BSV	Bluespec System Verilog
CPM	Computer Programming Manual
CRASH	Clean-slate design of Resilient, Adaptive, Secure Hosts
CW	ConcreteWare
DARPA	Defense Advanced Research Projects Agency
DC	Disjunction Category
FPGA	Field Programmable Gate Array
GC	Garbage Collection
IFC	Information Flow Control
I/O	Input/Output
ISA	Instruction Set Architecture
NaV	Not-a-Value
OS	Operating System
PAT	Principals Authorities and Tags
PC	Program Counter
PUMP	Programmable Unit for Metadata Processing
SAFE	Semantically Aware Foundation Environment
SKO	Safe Knowledge Online
TMU	Tag Management Unit
UART	Universal Asynchronous Receiver/Transmitter